НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»
ІНСТИТУТ ПРИКЛАДНОГО СИСТЕМНОГО АНАЛІЗУ
КАФЕДРА МАТЕМАТИЧНИХ МЕТОДІВ СИСТЕМНОГО АНАЛІЗУ

«До захисту допущено»
В. о. завідувача кафедри

О.Л.Тимощук

«___» _____20 __ р.

Дипломна робота

на здобуття ступеня бакалавра з напряму підготовки 6.040303 «Системний аналіз» на тему: «Синхронізація процесів у розподілених системах»

Виконав студент IV курсу, групи КА-54
Прозур Віталій Олександрович

_____ Керівник:
доцент кафедри ММСА, к.т.н., доц.
Коваленко А.Є. _____

Консультант з економічного розділу:
доцент кафедри теоретичної та прикладної економіки КПІ ім. Ігоря Сікорського, к.е.н., доц.
Шевчук О.А. _____

Консультант з нормоконтролю:
доцент кафедри ММСА, к.т.н., доц.
Коваленко А.Є. _____

Рецензент:
доцент кафедри системного аналізу та теорії прийняття рішень Київського національного університету імені Тараса Шевченка, к.ф.-м.н., доц.
Зінько П.М. _____

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань
Студент

_____

Київ
2019

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Інститут прикладного системного аналізу
Кафедра математичних методів системного аналізу

Рівень вищої освіти – перший (бакалаврський)

Напрям підготовки (програма професійного спрямування) – 6.040303

Системний аналіз (Системи і методи прийняття рішень)

ЗАТВЕРДЖУЮ

В.О.Завідувача кафедри

_____ О.Л. Тимощук

«___»_____20__ р.

ЗАВДАННЯ

на дипломну роботу студенту

Прозура Віталія Олександровича

1. Тема роботи «Синхронізація процесів розподілених систем», керівник роботи Коваленко Анатолій Єпіфанович к.т.н., доцент, затверджена наказом по університету від «__»____2019 р. №_____

2. Термін подання студентом роботи _____

3. Вихідні дані до роботи _____

_____

4. Зміст роботи _____

_____

_____

5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо) _____

_____

_____

6. Консультанти розділів роботи[1*]

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Економічний	Шевчук О.А., доцент		

7. Дата видачі завдання_____

1

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка

Студент В.О. Прозур

Керівник роботи О.Л. Тимощук

РЕФЕРАТ

Дипломна робота: 135 с., 7 табл., 24 рис., 6 дод., джерела.

РОЗПОДІЛЕНІ СИСТЕМИ, АЛГОРИТМИ СИНХРОНІЗАЦІЇ ПРОЦЕСІВ, ВЗАЄМОДІЯ ПРОЦЕСІВ, ВЕБ-СЕРВІС, КЛІЄНТСЬКИЙ ДОДАТОК

Об’єкт дослідження – принципи синхронізації процесів розподілених систем та способи їх реалізації.

Предметом дослідження є моделі процесів розподілених систем, та алгоритми синхронізації взаємодіючих процесів.

Актуальність роботи полягає в тому, що синхронізація процесів розподілених систем о основою для їх стабільного і надійного функціонування.

У роботі проведено дослідження існуючих алгоритмів синхронізації процесів розподілених систем на основі розподілених транзакцій і використання різноманітних механізмів керування процесами.

Розроблено систему програмної підтримки реалізації інтерфейсу для керування процесами з використанням розподілених транзакцій та інтерфейс клієнтського додатку для забезпечення зручної взаємодії компонентів системи.

ABSTRACT

Bachalorthesis: 135 p., 7 tabl., 24 fig., 6 append., sources.

The theme: Synchronization of distributed systems processes.

DISTRIBUTED SYSTEMS, ALGORITHMS OF PROCESSES SYNCHRONIZATION, PROCESSES INTERACTION, WEB SERVICES, CLIENT APPLICATIONS

Object of research - principles of processes synchronization in distributed systems and ways of their realization.

The subject of the research is the distributed systems processes model and interacting processes synchronization algorithms.

The urgency of the work is that distributed systems are a common thing and therefore the study of the problems of the processes interaction is an important task.

The research of existing algorithms of distributed systems processes synchronization is carried out. Distributed transaction system and client application for ergonomic interaction are developed.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

БД	–	база даних
МД	–	модель системного діагностування
ОС	–	операційна система
ПК	–	персональний комп’ютер
РІС	–	розподілена інформаційна система
IP	–	IP-protocol
RMON	–	remote monitor
SNMP	–	Simple network management protocol
TCP	–	transport comunication protocol
VPN	–	virtual private network
API	–	application programming interface

ВСТУП

Для створення розподілених комп’ютерних систем актуальною є синхронізація розподілених процесів. Розробка і впровадження алгоритмів синхронізації розподілених процесів покращує функціональні характеристики автономних систем.

В роботі проведено аналіз і наведено приклад релізації деяких існуючих моделей розподілених систем, розглянуто методологію застосування синхронізації процесів для реалізації процесів складних інформаційних систем.

Метою роботи було розглянути існуючі алгоритми синхронізації процесів в розподілених системах, реалізувати існуючий алгоритм та надати зручний користувацький інтерфейс.

Об’єкт дослідження – принципи синхронізації процесів розподілених систем та способи їх реалізації.

Предметом дослідження є моделі процесів розподілених систем, та алгоритми синхронізації взаємодіючих процесів.

Актуальність роботи полягає в тому, що розподілені системи посідають провідне місце у інформатизації суспільства. Реалізація таких систем суттєво залежить від створення програмного забезпечення для синхронізації розподілених процесів. Синхронізація процесів потребує керування транзакціями виконання груп процесів в умовах можливих відмов окремих процесів. В першому розділі розглянуто різні моделі розподілених систем, засоби та механізми, що використовуються для їх побудови.

В другому розділі вказані відомі алгоритми, що розв’язують проблеми синхронізації, показано приклади їх реалізації. В третьому розділі наведено приклад розподіленої системи та варіант реалізації алгоритму розподілених транзакцій. Також описано інтерфейс додатку для роботи даними, Четвертий розділ дозволяє проводиться функціонально-вартісний аналіз програм.

РОЗДІЛ 1 МОДЕЛЬ РОЗПОДІЛЕНОЇ СИСТЕМИ

1.1 Поняття розподіленої системи

Розподілена система являє собою сукупність незалежних комп'ютерів, які сприймаються користувачем єдиною об'єднаною системою. Розподілена інформаційна система (РІС) є розподіленою системою з єдиними інформаційними ресурсами. Доступ до інформаційних ресурсів регламентується встановленими правилами доступу до неї, процесами обробки, передавання, зберігання, перетворення й використання інформації [1].

Узагальнену організацію РІС для комп'ютерів А, В, С з локальними операційними системами (ЛОС) показано на рис. 1.1.

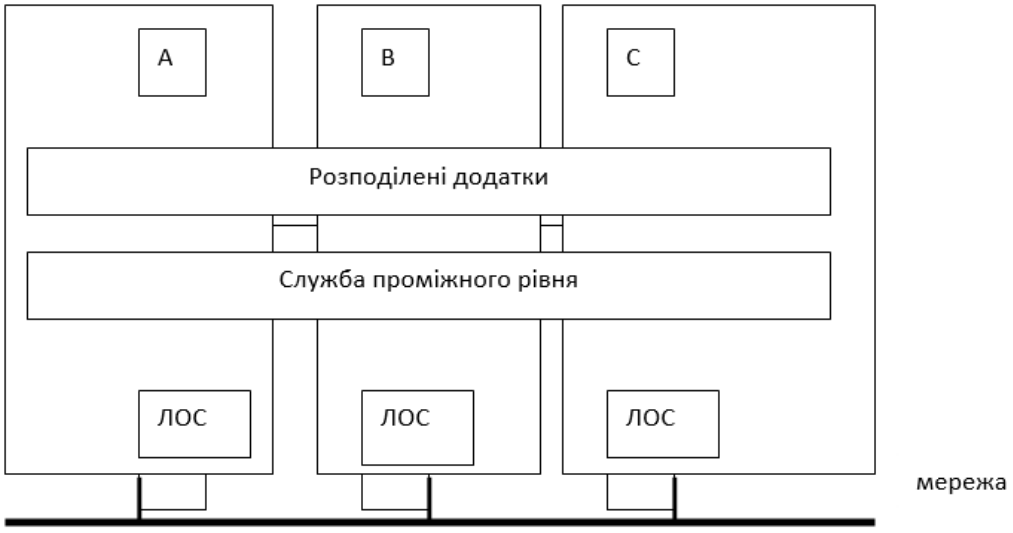


Рисунок 1.1 - Узагальнена організація РІС

Прикладний рівень розподілених додатків забезпечує єдине середовище взаємодії додатків. Служби проміжного рівня надає єдине середовище додаткових послуг при взаємодії з ОС окремих підсистем А, В, С.Така РІС підтримує прозорість системи для користувачів.

1.2 Основні задачі створення РІС

До основних задач створення РІС відносяться:

1. Організація з’єднання користувачів із ресурсами, яке спрощує обмін інформацією, зокрема, мультимедійною.

2. Забезпечення прозорості, яка полягає у прихованні для користувачів процесів і ресурсів на множині комп’ютерів.

3. Забезпечення відкритості РІС, яка передбачає наявність стандартних засобів або служб доступу до системи широкому колу користувачів.

4. Масштабованість, яка полягає у можливості розширення й модифікації РІС. Найбільш поширеними є масштабованість:

- за розмірами (підключення додаткових користувачів і ресурсів);
- географічна масштабованість (за місцеположенням);
- адміністративна (тобто взаємодія між собою адміністративно незалежних організацій).

Організація з’єднання у WAN передбачає підтримку віртуальних з’єднань за допомогою схем групової роботи. Віртуальні організації можуть включати десятки і сотні організацій, які розглядаються як члени однієї групи для розв’язання спільних задач із використанням систем групового доступу [1,2].

1.3 Програмне забезпечення проміжного рівня

Програмне забезпечення проміжного рівня дозволяє поєднати масштабованість і відкритість мережних ОС з прозорістю та простотою у використанні розподілених ОС. Таке поєднання дозволяє створювати РІС з такою загальною структурою, як показано на рис. 1.2.

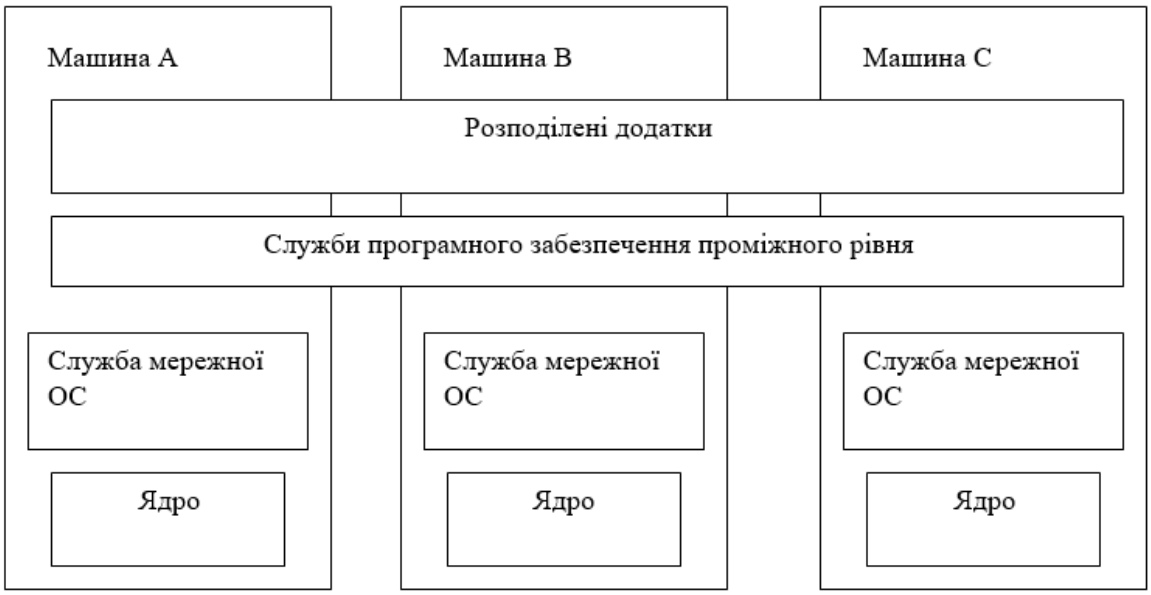


Рисунок 1.2 - Структура мільтикомп'ютерної системи із застосуванням проміжного рівня

Структура забезпечує поєднання можливостей окремих ОС і забезпечення прозорості, масштабності та відкритості системи. Основне завдання програмного забезпечення (ПЗ) проміжного рівня - приховати різноманіття базових платформ ОС від додатків [2].

Для розроблення ПЗ використовують деякі моделі або параметри. Основна частина ПЗ ґрунтується на моделі, яка визначає розподіленість і зв'язок. Найпростішими є моделі UNIX, коли всі ресурси системи (віддалені, локальні файли, принтери тощо) розглядаються як файли. Інша модель - це модель віддаленого виклику процедур RPC (RemoteProcedureCall).

1.4 Модель віддаленого виклику процедур

Мережний обмін приховується за рахунок виклику процедур RPC з віддалених машин. У разі виклику процедури параметри прозоро передаються у віддалену машину, яка цю процедуру виконує, а результат повертається до машини виклику (у випадку oneway може не повертатись).

Відповідно до об’єктного підходу застосовують розподілені об’єкти. Кожен з них реалізує інтерфейс, який приховує внутрішні деталі процесу об’єкта від користувача [3]. Інтерфейс реалізує методи, а процес бачить лише інтерфейс. Часто розподілені об’єкти розміщуються в одній машині, а доступ до його інтерфейсу відкривається з множини інших машин.

Щоб викликати процесором метод, інтерфейс перетворює його у повідомлення, яке надсилається об’єкту. Об’єкт виконує метод і повертає результат. Реалізація інтерфейсу перетворює повідомлення результату в значення, яке повертається і передається процесу. Так, у WWW використовується модель розподілених документів. Документ містить посилання на інші документи, тобто забезпечує їх зв’язок.

Основними службами проміжного рівня є:

1) засоби прозорого доступу до віддалених даних (файлових систем, розподілених БД, www);

2) служба віддаленого доступу, зокрема для виклику процедур і звертання до розподілених об’єктів;

3) служба іменувань. Наприклад, у WWW для цього застосовують ім′я URL;

4) засоби зберігання даних (засоби схоронності persistence), зокрема, розподілені файлові системи, інтегровані бази даних або засоби зв′язку додатків з базами даних;

5) засоби розподілених транзакцій, які здійснюють множину

операцій зчитування і записування в мережах однієї атомарної операції. Атомарна операція або виконується, або не виконується (неуспішне виконання). Дані транзакції можуть розміщуватись на різних машинах;

6) засоби розподіленого захисту, які є однією з найскладніших служб проміжного рівня. Ці засоби не можуть спиратись на локальні ОС.

Додатки зазвичай орієнтовані на певну РІС [3]. Неповнота інтерфейсу проміжного рівня з додатком примушує створювати спеціальний інтерфейс додатка з ОС платформи. А це погіршує відкритість проміжного рівня РІС і системи, оскільки потребує модифікації інтерфейсу у разі зміни платформи.

1.5 Рівнева структура розподіленої системи

З врахуванням цих особливостей еталонна модель мережної взаємодії з врахуванням проміжного рівня має структуру (рис. 1.3)

- фізичної реалізації;
- передавання даних;
- мережний рівень;
- транспортний рівень;
- проміжний рівень;
- рівень додатків.

Рисунок 1.3 – Структура моделі мережної взаємодії

1.6 Обмін даними за RPC

Віддалений виклик процедур RPC (RemoteProcedureCall) було вперше запропоновано у 1976 р. Сутність RPC полягає у можливості виклику процедури, що міститься в інших машинах [4]. Процес, запущений на машині А, викликає процедуру, розміщену у машині В. Виконання процедури в А призупиняється і виконується процедура на машині В.

Передавання інформації з процесу виклику до викликаної процедури відбувається через параметри і повертається до процесу у вигляді результату виконання процедури. Параметр може передаватись за значенням або за посиланням.

Наприклад, виклик з процесу у машині А віддаленої процедури read у машині В може мати такий вигляд мовою С:

count = read(fd, buf, bytes)

де fd – ціле число, яке є вказівником на файл;

buf – масив символів для зчитування даних;

bytes – кількість байтів, ціле число.

Передавання виклику на виконання процедури показано на рис. 3.6.

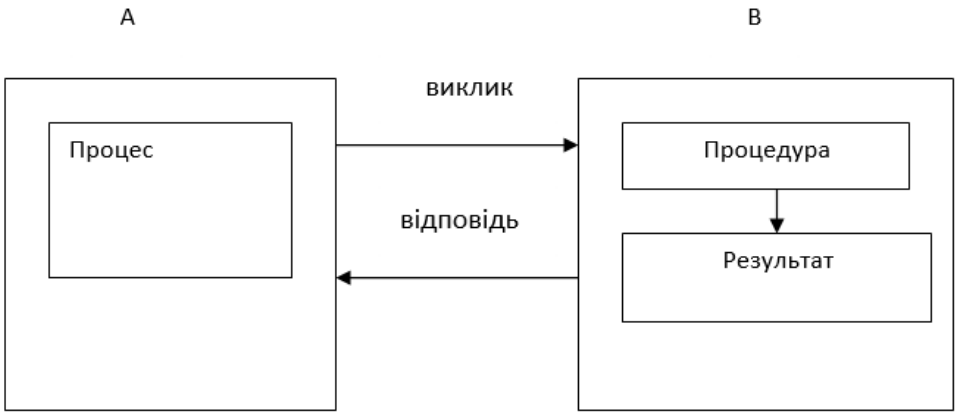


Рисунок 1.4 - Обмін повідомленнями між відправником і ортимувачем

Параметри − значення fd, bytes – безпосередньо копіюються у стек процедури, а параметр – посилання buf – дозволяє визначити початок масиву даних. За локальної реалізації процедури в межах однієї машини положення вказівника стека і розміщення даних до виконання виклику і під час виконання процедури, як показано на рис. 1.5.

Після виконання процедури система розміщує результат у регістр, вилучає адресу повернення і повертає керування у викличну програму. Викликана процедура використовує адресу масиву buf.

У деяких мовах застосовують виклик через копіювання та відновлення (call-by-copy/restore). У разі такого виклику відбувається копіювання в стек змінної, як за виклику за значенням. Після завершення виклику виконується копіювання цієї змінної зі стека з вилученням значення змінної [5].

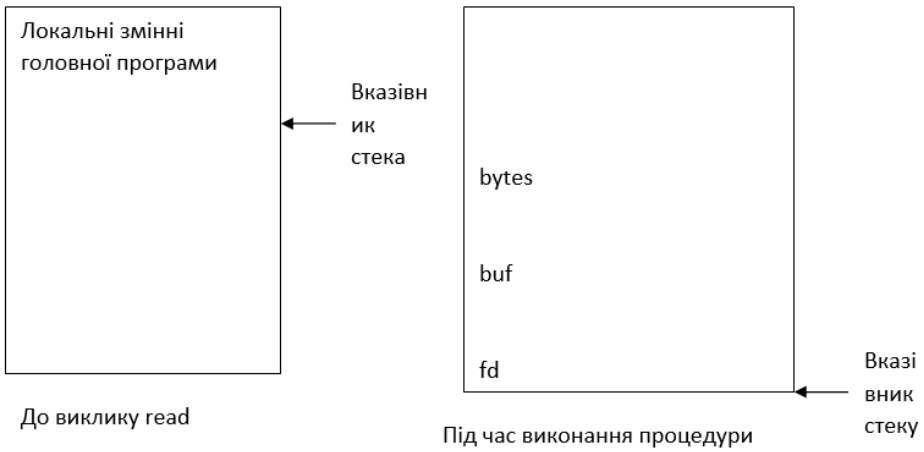


Рисунок 1.5 - Передавання параметрів запиту процедури read в RPC на одній машині

Прозорість команд RPC (зокрема, read) забезпечується за допомогою використання клієнтської заглушки, яка пакує параметри у повідомлення і надсилає процедурою send на сервер. Далі клієнтська заглушка викликає процедуру receive і блокується до отримання відповіді (рис. 1.6).

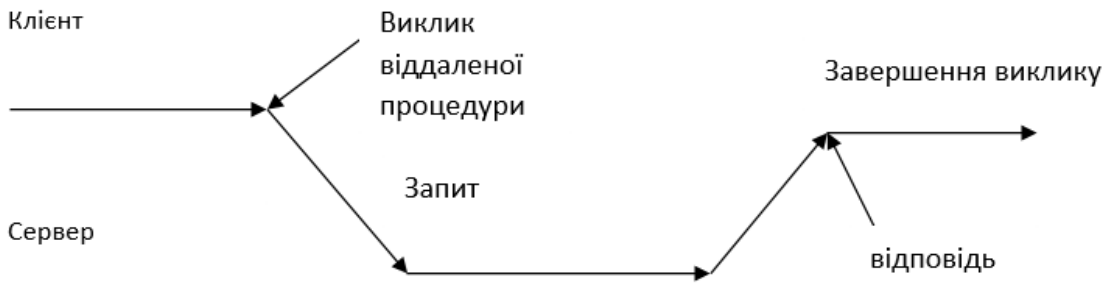


Рисунок 1.6 - Схема взаємодія клієнта і сервера у RPC

1.7 Послідовність використання заглушок клієнта і сервера

Повідомлення, отримане на сервері, операційною системою на сервері передається серверній заглушці (server stub), яка перетворює запит у виклик локальних процедур сервера.

Для отримання повідомлення заглушка запускає процедуру receive і блокується, очікуючи повідомлення. Після завершення оброблення виклику результат спаковується і процедурою send передається клієнту. Далі викликається процедура receive.

Операційна система клієнта визначає, що отримане повідомлення належить заглушці, копіює це повідомлення в буфер очікування, а клієнтський процес розблоковується. Клієнтська заглушка розпаковує повідомлення, копіює у пам′ять програми, яка її викликала, і завершує роботу, передаючи в неї код повернення. Ця програма отримує результат у буфері, тобто без визначення місця його походження.

Таким чином, клієнтська заглушка перетворює виклик процедури у локальний виклик процедури сервера (рис. 1.7).

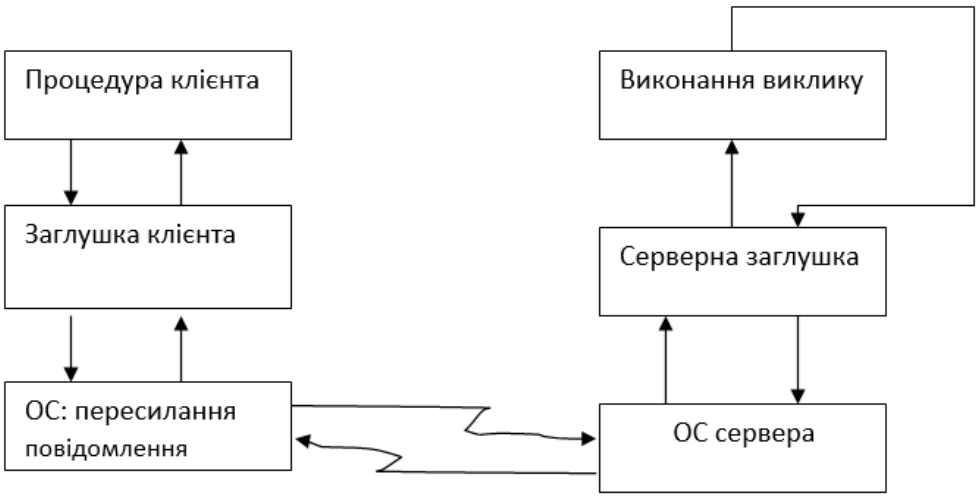


Рисунок 1.7 – Схема локального виклику процедури сервера

Існують деякі особливості передавання в RPC. Передавання за значенням спаковування параметрів (маршалінг параметрів – parameter marshaling) передбачає у повідомленні наявності імені віддаленої процедури, за яким на серверній заглушці, можливо із застосуванням коман типу switch, визначають саму процедуру [4,5].

Якщо використовуються параметри різних типів, можлива їх невідповідність у клієнта і сервера, зокрема за форматом і порядком слідування байтів. У разі пересилання даних за посиланням виникає проблема пересилання вказівника. Найкращим рішенням є заміна операції посилання на операцію копіювання/відновлення.

Для складних структур даних (графів, списків тощо) існують спроби передавання параметрів передаванням серверній частині реальних вказівників з подальшою генерацією кодів процедур для роботи з цими вказівниками.

Для спрощення побудови інтерфейсу в RPC можна застосовувати мову побудови інтерфейсів IDL (Interface Definition Language). Для побудови повідомлень клієнт і сервер повинні домовитись, за допомогою яких протоколів відбувається обмін повідомленнями (TCP/IP, UDP або інший) з урахуванням переваг і недоліків такого вибору.

1.8 Розширені моделі RPC

Для організації локальної взаємодії процесів на одній машині застосовують засоби між процесорної взаємодії IPC (InterProcessCommunication), які надають базові ОС. Так, в UNIX це засоби керування розподілюваною пам′яттю (shared memory), каналами, чергами повідомлень сумісного використання.

Засоби IPC переважно ефективніші за мережні засоби RPC на окремій машині. Компромісом для деяких ОС (наприклад, Spring, Solaris) є

застосування механізму, еквівалентного RPC, який має назву doors (двері або входи) і подібний механізм спрощеного виклику RPC (lightweight RPC). Принципи організації механізму показано на рис. 1.8.

Під час реєстрації входу сервером вхід отримує ідентифікатор, який можна використовувати як символічне ім’я входу. У Solaris виклик такого ідентифікатора з файловим іменем виконується простим запитом fattach. Запит door_call є системним і повертається через системний виклик door_return [6].

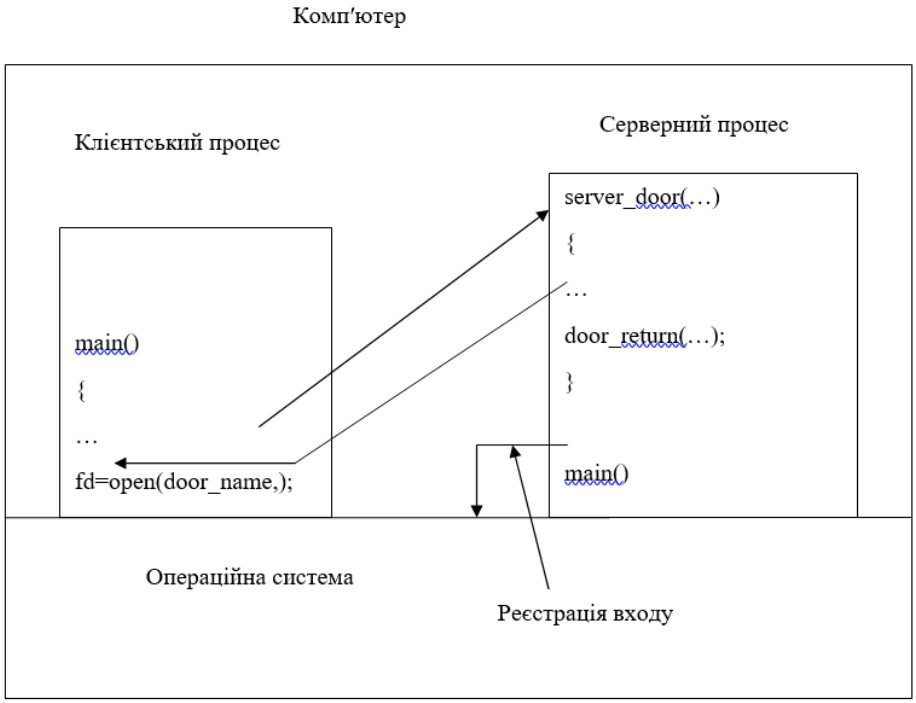


Рисунок 1.8 - Схема застосування механізмів door

Виклик процедур для віддаленої машини виконується за допомогою механізмів розширеного PRC. Стандартним є призупинення роботи клієнта на час виклику і отримання відповіді від сервера. Удосконаленим є асинхронний виклик RPC (asynchronous RPC), за яким клієнт продовжує роботу після виклику після отримання від сервера підтвердження про початок оброблення запиту. Удосконаленням є відкладений синхронний виклик RPC (deferred synchronous RPC), за яким сервер отримує від клієнта перелік потрібних для виконання запиту хостів (host), і підтверджує їх отримання. Існує також варіант одностороннього RPC, за яким клієнт не очікує відповіді від сервера.

1.9 Механізми віддаленої взаємодії

Механізм RPC застосовують як основу проміжного рівня. Найбільш відомими його варіантами є SunRPC та DCE (DistributedComputingEnvironment) RPC [9]. Версії DCERPC розроблено для UNIX, VMS і продуктів Microsoft.

Основними службами DCE RPC є:

1. Служби, що є частиною DCE:

– всесвітняслужбарозподіленихфайлів (distributedfileservice);

– службакаталогів (directoryservice) длявідслідковуваннямісцезнаходженняресурсусистеми;

– службазахисту (security serveice);

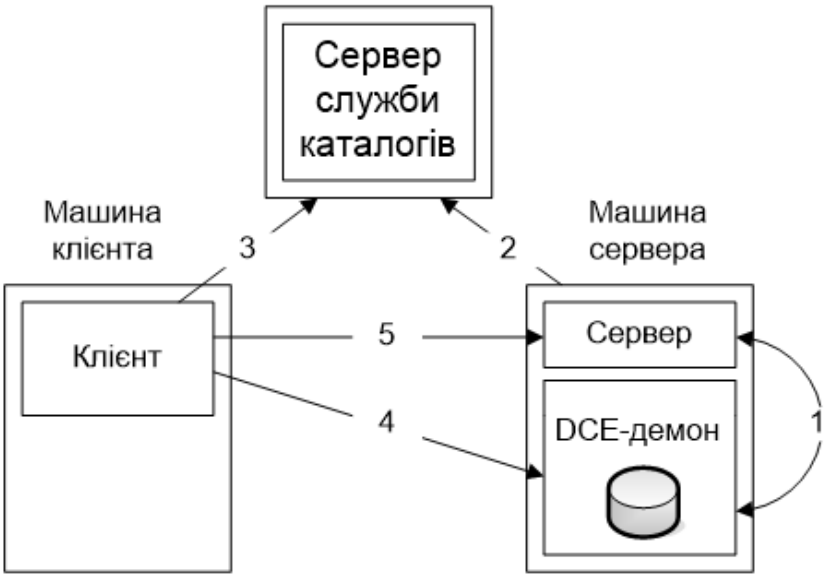
– службарозподіленогочасу (distributed time service).

2. Служби на основі додатків.

Системи RPC виконують такі процеси:

1) автоматичне визначення потрібного сервера і встановлення зв’язку між клієнтом і сервером через прив’язування (binding);
2) керування транспортуванням повідомлень:
3) автоматичне відслідковування перетворень. даних між клієнтом і сервером:
4) приховування реалізації взаємодії від користувача:
5) підтримання різноманіття мережних протоколів і подання даних.

Прив′язка клієнта до сервера виконується за схемою, показаною на рис. 1.9.



1 – реєстрація кінцевої точки; 2- реєстрація служби; 3-пошук сервера служби каталогів; 4- запит кінцевої точки; 5 – виконання виклику RPC

Рисунок 1.9 - Схема прив’язування клієнта до сервера DCE

Клієнт до сервера прив’язується через реєстрацію клієнта на машині каталогів з подальшим використанням відповідного сервера і кінцевої точки або порту сервера [1-3]. При цьому на машині сервера здійснюється фіксація сервера і відповідної кінцевої точки (порту) у спеціальній БД за допомогою DCE-демона. На підставі вихідних даних сервера, кінцевої точки (порту) і визначеного протоколу відбувається взаємодія клієнта із сервером.

Принципи RPC можутьзастосовуватись для об′єктно-орієнтованих технологій на основі розподілених віддалених об′єктів за схемою заміщував клієнта (рис. 1.10).

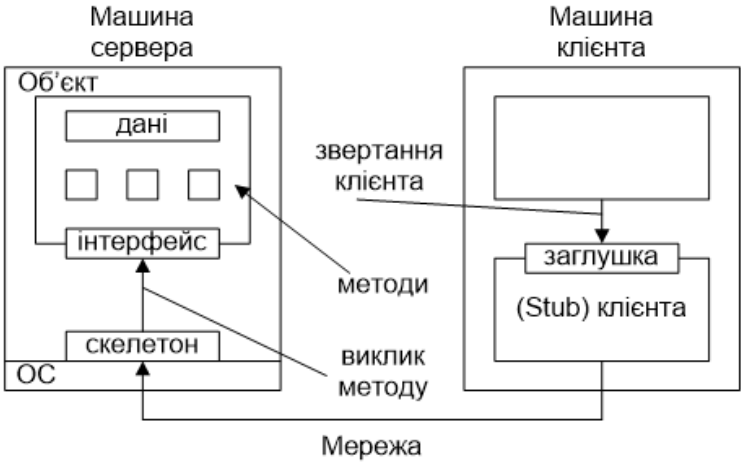


Рисунок 1.10 – Схема організації RPC для розподілених об’єктів

Звертання клієнта через замісника або представника (proxy) називають методом віддаленого звернення до методу RMI (RemoteMethodInvocation). Інтерфейс об’єктів RMI пишуть мовою IDL, Java та інших засобах, за умови, що інтерфейси об’єктів під час розроблення клієнтського додатка відомі, а в разі зміни клієнтського інтерфейсу виконується перекомпіляція.

Передавання параметрів у RMI для розподілених об’єктів розрізняється залежно від того, чи є об’єкт локальним або віддаленим. Для локальних об’єктів у клієнтській машині, де є посилання, об’єкт копіюється повністю і в процесі копіювання передаєтьсяклієнту.

Розрізняють схоронний і транзитний зворотні зв’язки [3,4,5]. У разі схоронного зв’язку за допомогою повідомлень додаток після відправлення повідомлення може завершити роботу. У разі транзитного зв’язку повідомлення зберігаються у системі лише на час роботи додатків. Зв’язок також поділяють на синхронний та асинхронний.

Висновки до розділу 1

Розглянуто узагальнену рівневу протокольну модель розподіленої системи. Визначено основні задачі, які ставляться при забезпеченні взаємодії компонентів системи на основі протоколів RPC.

Проведено аналіз механізмів реалізації взаємодії між її елементами та процесами. Модель виклику процедур РРСдозволяє отримувати доступ до віддалених даних та обмінюватись даними між різними машинами.

Проведено аналіз розширених моделей віддаленої взаємодії для протоколів RPC, RMI, які застосовуються у Unix – подібних розподілених системах

Визначено основні механізми віддаленої взаємодії у розподілених системах та синхронізація процесів взаємодіючих процесів.

РОЗДІЛ 2 ЗАСОБИ СИНХРОНІЗАЦІЇ

2.1 Моделі синхронізації процесів

Узгодження взаємодійних процесів відбувається застосування різних синхронізаційних принципів під час роботи процесів. Проблеми взаємодії двох або більше процесів описуються задачами синхронізації, типовими з яких такі:

- взаємовиключення;
- «виробники-споживачі»;
- «письменники - читачі»;
- «філософи за обідом».

Якщо ігнорувати необхідність синхронізації, як наслідок можуть виникнути до тупики або дедлоки, які унеможливлюють реалізацію декільнох процесів. Для того щоб уникнути загрози дедлоків використовуть алгоритмисихронізації розроблені спеціально для подолання можливих дедлоків [6-12].

Управління процесами передбачає реалізацію певних операцій, серед яких є:

–створення процесу;

–запуск (вибір) процесу.

–знищення процесу;

–призупинення процесу;

–пробудження процесу;

–блокування процесу;

–зміна пріоритету процесу;

–відновлення процесу;

Створення процесу складається з видачі імені процесу, додаванняв список імен процесів, визначенні пріоритетності виконання цих процесів,

реалізація блоку керування процесом, виділення ресурсір та інше. Процес має можливість створити дочірній процес, отримуючи статус батьківського процесу.

Знищення процесу - вилученння даних про процеси, в тому числі у видаленні імен, доступів до використовуваних ресурсів.

Відновлення процесу передбачає відтворення даних, що потрібні для реалізації даного процесу:адреси стека, дані лічильника команд, стани регістрів процесора та інше. Для реалізації цих функційвикористовуютьтак звані «зрізи» стану ходу процесу – контрольні точки.

2.2 Взаємне виключення з активним очікуванням

Для паралельної одночасної роботи процесів та ефективного застосуванняспільних ресурсів для процесівнеобхіднозадовольнити чотирьом умовам:

1) у критичній області не може перебувати більше одного процесу одночасно;

2) у програмі не можуть використовуватися допущення про швидкодію чи кількість процесів;

3) процес, що знаходяться в критичній області не можуть бути заблоковані процесами поза нею;

4) не можна допустити ситуацію, де процес нескінченно очікує потрапляння до критичної області.

Для контролю перебування процесу в критичній зоні використовується активне очікування іншого процесу.

Нижче перелічено основні способи взаємного виключеннян з ціллю запобігання втручання в критичну область при наявному там процесу:

1) заборонені переривання. При входженні процесу в критичну зону, переривання повністю забороняються, в тому числі й переривання за таймером;

2) використання змінних блокування. Використовується одна загальна змінна для всіх процесів. Спочатку вона рівна нулю або іншому значенню за замовчуванням, а у випадку входження процесу в критичну область значення змінюється;

3) строге чергування.

2.3 Алгоритм Петерсона

Алгоритм Петерсона складається з двох процедур мовою ANSIC: процедури in_reg, що реалізовує вхід в критичну область і процедури out_reg виходу з критичної області [3]. Приклад таких процедур процедур для двох процесів може мати такий вигляд:

```
#defineFALSE 0
#define TRUE 1
#define M 2
int trn;
int intrstd[M];
void  in_reg(int proc)
{
int  othr;
othr=1-proc;
intrstd[proc]=TRUE;
trn = proc;
while (trn==proc && intrstd[othr]==TRUE )
}
void  out_reg(int proc)
```

{intrstd[proc]=FALSE; }

Процесом 1 або 0викликається відповідна своєму номеру процедура, що в свою чергу входитьукритичну зону in_reg(int proc) ічерез intrstd[proc]=TRUE фіксується присутністьуцій зоні.

Якщо дана область вже зайнята іншим процесом (1 або 0), то даний процес починає виконувати цикл while, таким чином очікуючи звільнення області [2, 3].

Після того як процес виконує необхідні дії вінзалишає область використовуючи процедуру out_reg(int proc) за допомогоюприсвоєння значення intrstd[proc]=FALSE.

При виклику процедуриin_reg двома процесами приблизно в один момент, змінній turn присвоюється значення більш пізнього процесу, і вінне вийде з циклу while і буде очікувати поки звільниться критична область.

2.4 Міжпроцесна взаємодія в одному адресному просторі

Активне очікування процесу призводить до неефективного використання машинного часу. Щоб уникнення цього застосовують спеціальні примітиви між процесорної взаємодії. Найпростіші з них пара примітивів sleep, wakeup, більш складні мютекси, семафори і монітори.

Примітив sleep - системним запитом, що блокує процес, який викликається. Wakeup має параметр, а саме процес, щотреба запустити. Щоб узгодити запити очікування та запиту для обох процесів використовується єдина адреса пам`яті. Примітиви sleep, wakeup застососвують для розв'язання різних проблем, наприклад проблеми “виробники-споживачі” (чи проблем обмеженого буфера).

При проблемі “виробники – споживачі” у випадку лише одного виробника та споживача першийставить елемент даних до буфера, підвищуючи значення лічильника cnt на один, тобто cnt += 1. Споживач же читає дані, змушуючи cnt на один, тобто cnt -= 1.

При значенні cnt = M (тобто при значенні в максимум кількості елементів, що може зберегти буфер), споживач починає очікування. Якщо значення cnt = 0 (тобто після вилучення останнього елементу (таким елементом вважається елемент при cnt = 1)) споживач починає очікування, оскільки йому вже нічого зчитувати.

Споживач може прокидатися (наприклад, при пробудженні його виробником), якщо cnt> 0, а виробник може пробуджуватись (наприклад, споживачем) у випадку, якщо значення cnt = M-1, оскільки при цьому звільниться достатньо місця для запису даних. Мається на увазі, що кожен з процесів (як виробник, так і споживач) вирішує завдання по активізації іншого [1-3].

Можливим розв’язком задачі “виробники - споживачі” є використаннянаступного програмного коду для виробника і споживача:

```
#define M=100
Int cnt =0;
void prod(void) {
int itm
while(TRUE) {
ite=prod_itm();
if (cnt==M) sleep();
ins_itm;
cnt+= 1;
if (cnt==1) wakeup(cons);
}
}
void cons(void)
{
```

```
int itm;
while (TRUE)
{
if (cnt==0) sleep();
itm = rem_itm();
cnt -= 1;
if (cnt==M-1) wakeup(prod);
cons_itm(itm);
}
}
```

Перегони при реалізації процесів можуть виникнути в той момент момент, коли споживач щойно з’яв елемент даних, а виробник відправив елемент до буферу і встановив сигнал пробудження, іншими словами активізацію споживача, який цей сигнал невикориставі перейшов у режим очікування. Утрачений сигнал активізації спричиняє перехід до очікування виробника. Виходить так що обидва процеси знаходяться в стані очування.

Семафор – це така змінна, що може набувати значення 0 (при відсутності сигналів активізації) або числа, що в свою чергу рівне кількості відкладених активованих сигналів. Над семафором виконують операції down (або P), up (або V), що є подібними до операцій sleep та wakeup. Down зменшує семафор на одиницю, при умові що той не рівний нулю, і повертається керування процесу.

За нульового значення семафора down не повертає керування процесу, і процес переводиться у режим очікування. Операції перевірки значення семафора, його зміна і переведення процесу у стан очікування виконуються як єдина елементарна дія.

Операція up збільшує значення семафора. За наявності кількох процесів, пов’язаних із семафором, що не мають можливості закінчити операцію down,

система при up вибирає один з цих процесів, зменшуючи кількость процесів на один, при цьому залишаючи значення семафора таким самим [4-6].

Мютекс (mutex – mutualexclusion) – це змінна, що може перебувати у двох наступних станах: “блоковано” (1) та неблоковано (0). Мютекс, по факту, спрощена версія семафора. Мютексу можна встановлювати значення за допомогою процедур mtx_lck, mtx_unlck. При входженні процесу або потоку до критичної області викликається процедура mtx_lck. Запит виконується і процес потрапляє у критичну область лише за умови, що мютекс не заблоковано. У випадку коли мютекс заблокований іншим процесом, то потік (чи процес), що його викликає, блокується і очікує коли потік що зараз знаходиться в критичній обсталі залишить її і викличе mtx_unlck [3]. У випадку коли мютекс блокує кілька потоків, то наступний буду обиратися випадковим чином.

Монітор являє собою набір різних змінних, процедур, та інших структур даних, які об`єднано в особливі модулі або пакети. Процесами можуть викликатися процедури монітора, але будь-яка процедура, якубуло оголошено не в моніторі, не може використовувати внутрішніх структур даних монітора, принаймні напряму.

При зверненні до монітора активних процесів має бути не більше одного. Блокування процесів може бути виконано за допомогою змінних станів та двох операцій - wait та signal [10,11]. У разі, якщо процедура монітора виявила власнунездатність продовжувати роботу (в тому числі, якщо виробник оголошує, що буфер вже заповнений), то вона починає виконувати операцію wait при будь -якій змінній стану, як варіант full. Це блокує викликаний процес і дає можливість іншим процесам (наприклад, споживачам) увійти до монітору. Процес має можливість активізовувати процеси, що очікують, наприклад, за допомогою встановлення операції signal за змінною стану, якою його було блоковано.

Розрізняють кілька підходів до розв’язання проблеми подальшого запуску процесу, з яких найбільш поширені запуск пробудженого процесу і

зупинка другого процесу (Хоар); метод, за яким процес, який виконував signal, має відразу ж залигити монітор (Бринч Хансен); комбінований метод, що дозволяє продовжувати роботу процесу, який виконував signal, і запускати процес, що чекає, лише після залишення монітору першим процесом.

2.5 Розв’язання проблеми синхронізації на основі примітивів взаємодії

У випадкузастосування семафорів є можливість застосовування трьох семафорів: для розрахунку кількості сегментів пам'яті буфера, що заповнено (full), кількості порожніх сегментів (empty) і уникненняодночасного доступу до буфера «виробника та споживача» (тобто контролюючи доступ до критичної зони) mutex. Програмний код для виробника і споживача на мові програмування С може мати наступний вигляд:

```
#defineM=1000 /* кількість сегментів буфера */
typedefintsemaph;
semaph mtx = 1;
semaph emp = M;
semaph fll = 0;
void prod(void)
{
 int itm;
 while(TRUE){
 itm=prod_itm();                    /* створення елемента даних */
 dwn(&emp);                         /* зменшити лічильник порожніх сегментів */
 dwn(&mtx);                         /* входження до критичної області */
 Ins_itm(itm);                      /* розмістити елемент даних у буфер */
 up(&mtx);                          /* вихід з критичної області */
 up(&fll);                          /* збільшити лічильник повних сегментів буфера
                                    */
```

```
}
}
void cons(void)
{
 int itm;
 while(TRUE){
 dwn(&fll);                    /* зменшити кількість порожніх сегментів
                               буфера
 dwn(&mtx);                    /* входження до критичної області */
 itm=rem_itm();                /* вилучити елемент даних з буфера */
 up(&mtx);                     /* вихід з критичної області */
 up(&emp);                     /* збільшити лічильник повних сегментів
                               буфера */
 cons_itm(itm);                /* обробка  елемента */
}
}
```

Взаємневиключеннянреалізуєтьсязвикористанням семафора mutex виконаннямкожнимпроцесомоперації down під час входженнядокритичної зони таоперації up на виходізкритичної зони. Семафори fll, emp використовуються для реалізації синхронізаціїпроцесів [6,7].

Справжня реалізаціямютексів у ОСпередбачає собою набагато ширшийнабіркоманд, якіпідіймають ефективністькеруванняпотоками.

Примітивимонітори можуть мати різні програмні реалізації. Наприклад, реалізація монітора для процедур виробника і споживача може мати такий вигляд:

```
mntr        exmpl
            int j;
            condition con;

            procedure prod();
            …
```

```
end;

procedure cos();
…
end;
end mntr;
```

При зверненні до монітора активних процесів не може бути більше одного. Особливості обробки виклику процедур моніторів враховуються компілятором, в тому числі перевіряються наявності активних процесів в системах з реалізацією взаємного виключення [3]. Потрібно також блокувати процеси, що не можуть продовжити виконання залежно від розв'язання проблеми взаємодії процесів.

Так для задачі “виробники-споживачі” є можливістьвикористати буфер, операції wait, signal, змінні стану.

Приклад розв’язку цієї проблеми з застосуванням моніторів має мати наступний вигляд [1]

```
mntr        ProdCons
            condition fll, emp;
            integer cnt;
            procedure ins(itm: integer);
            begin
            if cnt=M then wait(fll);
            ins_itm(itm);
            cnt:=cnt+1;
            if cnt=1 then signal(emp)
            end;

            function rem: integer;
            Begin
            if cnt=0 then wait(emp);
            rem:=rem_itm;
```

```
                cnt:=cnt-1;
                if cnt=M-1 then signal(fll)
                end;
                cnt:=0;
end             mntr;

                procedure prod;
                begin
                while true do
                Begin
                itm:=prod_itm;
                ProdCons.ins(itm)
                End
                end;

                procedure cons();
                begin
                while true do
                begin
                itm:= ProdCons.rem;
                cons_ite(itm)
                end
                end;
```

Якщо процесвиробника перебуває в моніторі, то керування споживачу може передатись не раніше завершення операції wait, після чого він може увійти в монітор.

2.6 Міжпроцесна взаємодія на основі повідомлень

Примітиви семафори, монітори не пристосовані для використання у розподілених системах. Для цих систем застосовують примітиви міжпроцесорної взаємодії на підставі повідомлень snd, rcv, які частододають до бібліотек ОС.

Узагальнений формат форматцихкоманд:

```
snd(dst, &msg);
rcv(src, &msg);
```

Параметри dst та src відповідають замісцепризначеннятаджерело, a&msg відповідає зазмістповідомленнь.

У разі якщонемає повідомлення у rcv, запит буде заблоковано або сформовано код помилки. У випадку, коли повідомлення надіслано повторно у текст повідомлення додається номер копії.

Реалізаціязадачі “виробники-споживачі” у випадку розподілених систем має мати такий вигляд на мові програмування С:

```
#define M 1000                      /* к-ть у буфері  */
void prod(void) {
int itm;                            /*  буфер повідомлення  */
msg m;                              /*  змінна повідомлення  */
while(TRUE) {
itm=prod_itm();
rcv(cons, &m);                      /*очікування повідомлення*/
bld_msg(&m, itm);                   /* повідомлення для відправлення*/
snd(consumer, &m);                  /*  відправити споживачу  */
}
```

```
}

void cons(void)
{
int itm, i;
msg m;
for (i=0; i<M: i++) snd(producer,        /*  відправити N повідомлень */
&m);
while(TRUE) {
rcv(prod, &m);                           /*  отримання даних  повідомлення   */

itm=extrct_itm(&m);                      /* елемент з повідомлення  */
snd(producer, &m);                       /*  відправити порожнє*/
cnsme_itm(itm);                          /*  обробити дані */
}
}
```

Виробник спочатку направляє N порожніх повідомлень (без елементів даних) споживачу. За наявності повідомлення виробник отримує порожнє повідомлення, але надсилає повне повідомлення з даними. Мається на увазі, що кількість повідомлень є постійною у системі [13, 14].При швидкісті виробника більшій,ніж швидкість споживача, затримуються всі повідомлення, а сам виробник буде заблокований. У разі якщо споживач працюватиме швидше за виробника, то у випадку якщо всі повідомлення порожні він блокується і очікує на не порожні повідомлень.

2.7 Обробка взаємоблокувань

Натепер не існує універсальних підходів до оброблення взаємоблокувань в ОС. Виділяють наступні підходи дляусунення впливу взаємоблокувань :

1) Структурне заперечення однієї з умов виникнення взаємоблокувань;

2) динамічне запобігання тупикови ситуаціям;

3) виявлення взаємоблокувань.

Виділяють різні виявлення взаємоблокувань за наявного одного або кількох ресурсів кожног типу. Основа алгоритмів - це розшук замкненого циклу для заданих вузлів графу Холта.

Одним з неоптимальних, проте одночасно й найпростіших алгоритмів пошуку циклу являється виконаннятаких п′яти кроків для кожного вузла графу:

1) Визначення списку L вузлів з вихідними не маркованими ребрами.

2) Додавання поточний вузла до кінця списку L. Перевірка на наявність іншого такого вузла у списку. У вападку, коли такий вузол можна отримати,маємо циклічний шлях, щодемонструє наявністі взаємоблокувань. Завершення виконання алгоритму.

3) При існуванні хоча б одного немаркованого ребра з теперішнього вузла, то переходимо до наступного кроку. Інакше перейдемо відразу до 5 кроку.

4) Обираємо довільне ребро, що ще не марковано, для даного вузла, та маркуємо і переходимо по ребру до наступного суміжного вузла. Переходимо до кроку номер 2.

5) Маємо тупик. Вилучаємо останній вузол та повертаємось до попереднього вузла, що був поточним. Позначаємо обраний вузол як поточний і повертаймось до кроку номер 2. У разі що це початковий вузол, то граф не міститить жодного циклу й алгоритм можна завершувати.

Цей алгоритм можна назвати алгоритмом повного перебору, щомає в основі маркування ребер шляху та виявлення замкненого шляху (циклу), що має початок і закінчення у деякому вузлі.

Знайдення взаємоблокувань - перший кроком до усунення його впливу, що він завдає комп′ютеру. Виділяють наступні підходи для відновлення після виявлення тупика такі:

1. За допомогою примусового вивантаження ресурсу, тобто вилучається від поточного власника і передається ресурсу іншого процесу. При такому вилученні його коректність як і подальше відновлення дуже часто залежить від типу процесу.

2. Відновлення через повернення, щоскладається з повернення до створеного раніше «зрізу» стану (контрольної точки). Контролбна точкадає змогу зберігати стан процесу в певний момент часу, в тому числі образ пам′яті, дані про ресурси, стан ресурсів та інше. Повернення до попереднього зрізу може спричинити потреби в доступі до ресурсів.

3. Відновлення через знищення процесів, сенс якого у вилученні процесів, які спричиняють взаємоблокування, або вилученні процесів, щозаймалють ресурси необхідні для розблокування. При можливості знищують перш за все ті процеси за виконання яких можна розпочати з початку.

2.8 Алгоритми синхронізації розподілених систем

2.8.1 Алгоритми синхронізації годинників

Взаємодія процесів у розподіленій системі можлива лише за умови їх синхронізації в часі при пересиланні, отриманні і передачі інформації.

Глобальна синхронізація годинників машин дозволяє проводити упорядкування ресурсів системи (версій файлів, повідомлень тощо). Годинники комп`ютерів не можуть забезпечити глобальну синхронізацію, оскільки мають похибки відліку, пов`язані із застосуванням кварцевих годинників. З часом їх використання призводить до розсинхронізації (clockskew). Тому час, який асоціюється з файлом, об`єктом повідомлення призводить до неправильної роботи програм, які з ними працюють [1,4,8].

В сучасних розподілених системах застосовують універсальний узгоджений час UTC (UniversalCoordinatedTime), який по суті замінив час за Грінвичем (Greenwichmeantime). Відлік UTC грунтується на глобальному часі за атомним годинником TAI (InternationalAtomicTime), який визначають як середній час тиків годинника на цезії -133 після 1.01.1959 р., поділений на число 9192631770.

Час UTC передається кількома радіостанціями MSF з Rugly (Велика Британія), супутникової системи GEOS. Годинник має таймер. Таймер містить лічильник (counter) на віднімання одиниці, і регістр (holdingregister). Таймер клієнта розраховано на формування сигналу переривання при досягненні нуля. Таке переривання називають тиком таймера (clocktick). Дата і час записані у кількості тиків. Існують централізовані алгоритми (наприклад, Крістіана, Берклі) і децентралізовані алгоритми встановлення годинників машин.

2.8.2 Алгоритми синхронізації Крістіана

Алгоритм Крістіана використовує машину приймача сигналу WWV, яку називають сервером часу (timeserver). Кожна з машин клієнта (активного) надсилає запит на цей сервер не рідше ніж $\delta/2\rho$ с., (де δ є припустима похибка, а ρ – константа максимальної швидкості дрейфу годинника (maximumdrifttime)), і отримує поточний час CUTC . На Рис.2.1 показано в часі взаємодію сервера і клієнта відліку часу.

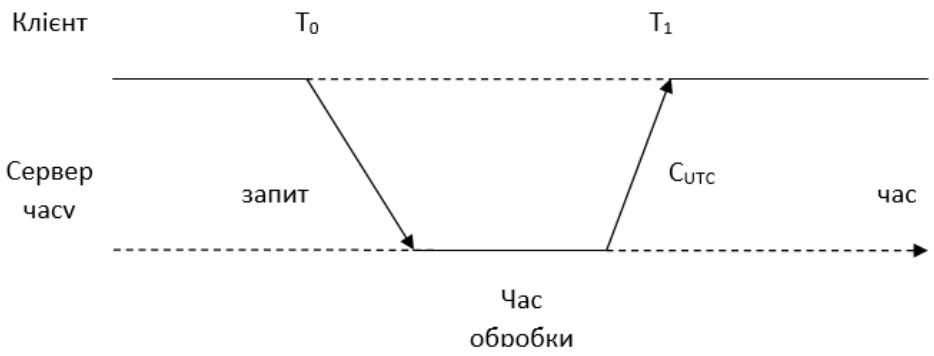


Рисунок2.1 - Взаємодія сервера і клієнта в часі

Управління клієнтом власного часу C при отриманні CUTC відбувається поступово, з використанням власного таймера. Таймер формує, наприклад, 100 переривань на секунду. При C < CUTC замість додавання 10 мс в таймер при перериванні додається лише 9 мс [14].

Позначаємо час обробки переривання I, а час проходження повідомлення (T1 – T0)/2, на яке необхідно збільшити час у повідомленні, при відсутності даних про I. При відомому часі I збільшення становить (T1 – T0 - I)/2. Оскільки час від A до B, і від B до A може біти різним, то виконується серія вимірів часу з подальшим отриманням середнього значення. Значення T1 – T0 , які перевищують порогове, відкидаються.

2.8.3 Алгоритми синхронізації Берклі

В алгоритмі Берклі сервер часу без WWV є активним демоном часу, який час від часу запитує кожну з машин про її час, обчислює середнє значення і пропонує перевести до нього всі машини шляхом відповідно корегування [3, 10-14].

Один з класів децентралізованих алгоритмів синхронізації будується на основі поділу часу на синхронізуючі інтервали фіксованої тривалості. Кожний

j-й інтервал розпочинається у момент $T_0 + jR$ та триває до $T_0 + (j+1)R$, T_0 являє узгоджений раніше момент часу, R – параметр системи.

На початку інтервалу кожна машина здійснює широкомовну розсилку значення поточного часу за своїм годинником, запускає локальний таймер, і починає збирати всі інші широкомовні пакети протягом часу S. За широкомовними пакетами, що було отримано, розраховується новий час. До прикладу, середнє значення (можливо з відкиданням крайніх значень).

До відомих алгоритмів цього класу відносяться протокол мережного часу NTP (Network Time Protocol), який забезпечує точність 1-50 мс. Останніми роками починають використовуватись алгоритми в межах Інтернет для створення синхронізованих годинників.

2.8.4 Алгоритми синхронізації логічних годинників

В багатьох випадках при взаємодії процесів застосовують узгодження, не обов`язково точне, годинників за часом, який називають логічним часом (logicalclock). Для синхронізації логічних годинників Лампорт визначив відношення a→b (“a відбувається раніше за b”). Відношення виконується, коли

1) a і b є подіями одного процесу, подія a відбувається раніше за подію b, і відношення a→b є істинним;

2) якщо a – подія відправки повідомлення одним процесом, а подія b - отримання його іншим, то відношення a→b є істинним. Повідомлення не можна отримати раніше, ніж воно було відправлено.

З транзитивності відношення випливає, що з відношень a→b, b →c, випливає відношення a→c.

Дві події x, y паралельні (concurrent), якщо вони відбуваються у різних процесах і відношення x→y, y→x не є істинними.

Для виміру часу події x застосовують відліки часу C(x). Для істинного a→b виконується C(a) <C(b), а для різних процесів при відправці повідомлень (подія a) і отриманні повідомленні (подія b) необхідно з'ясувати C(a)<C(b). Час тільки збільшується, тобто корекція робиться лише в сторону збільшення.

2.8.5 Алгоритми синхронізації Лампорта

За алгоритмом Лампорта [1,3] кожне повідомлення містить час відправки за годинником відправника. При отриманні повідомлення годинник відправника є дійсним, інакше відбувається корегування його годинника на +1. Для того, щоб для різних подій a, b, C(a) <>C(b) використовують додавання десяткового розряду.

Для підтримки причинно-наслідкових зв`язків застосовують векторні відліки (відмітки) часу (vector time stamps), за якими кожному процесу Pi приписують вектор Vi з властивостями:

Vi[i] – кількість подій, які відбулися з Pi до даного моменту часу;

2)якщо Vi[j] = K, то процес Pi знає, що з процесом Pj відбулося K подій.

2.9 Глобальний стан.

Глобальний стан (global state) розподіленої системи – це стан, який включає локальний стан процесу з повідомленнями, що перебувають у русі (тобто відправлені, але не доставлені). Локальний стан процесу залежить від його аналізу і використання. В якості глобального стану часто застосовують

розподілений знімок (distributed snapshot), властивістю якого є відображення несуперечливого глобального стану [3,4].

Знімок містить записи відправки і отримання повідомлення, або лише відправки. Алгоритм побудови знімка може ініціюватись кількома процесами і тому можуть створюватись кілька знімків станів. Наприклад, процес Pi починає з запису власного локального стану і відправляє маркер кожним із своїх вихідних каналів. Після отримання маркера, процеси фіксують свої локальні стани і направляють процесу, що ініціював створення знімка.

2.10 Алгоритми голосування.

Алгоритми голосування застосовують для визначення процеса – координатора, наприклад, з найбільшим номером, з групи взаємодіючих процесів. Поширеним є алгоритми задираки і кільцевий алгоритм.

Алгоритм задираки (bully algorithm) передбачає ініціалізацію процесом P голосування при відсутності відповіді на запити від координатора за такими правилами.

1)Процес P надсилає процесам, у яких більше ніж у нього номери, повідомлення ГОЛОСУВАННЯ.

2)Якщо ніхто не відповідає, то процес P стає координатором.

Якщо відповідає процес з більшим номером, то він стає координатором і P припиняє голосування. Процес завершується визначенням єдиного координатора для групи процесів. Якщо процес перебував у неробочому стані і почав працювати, то він організує голосування. Наприклад, як показано на Рисунок 2.2.

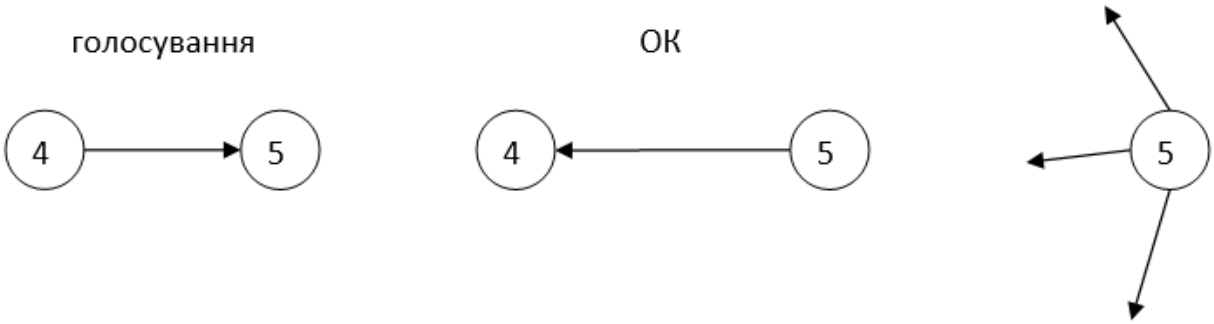


Рисунок 2.2 – Схема процесу голосування

Кільцевий алгоритм передбачає упорядкованість процесів, деусі процеси знають, хто їх нащадки. При відсутності відповіді від координатора, процес Р направляє повідомлення ГОЛОСУВАННЯ нащадку з власним номером процесу. Коли нащадок не відпрацьовує, то надсилається повідомлення наступному члену кільця, додаючи свій номер процесу до списку номерів у повідомленні (Рисунок 2.3).

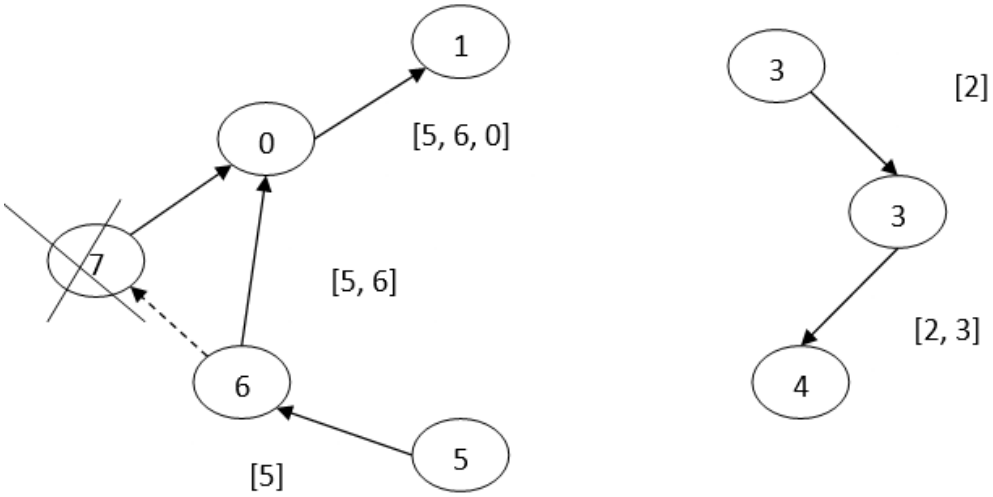


Рисунок 2.3 – Схема процесу голосування

2.11 Взаємне виключення розподілених процесів

Взаємовиключення це відсутность доступу до інших процесів у області, яка критична. Системи з великою масою процесів легше програмувати за допомогою критичних зон. Якщо настає момент, коли процесу необхідно зчитувати або обновлятиспільно використовувані структури даних,цей процес входить в критичну область, щоб потім переконатися, що більше не один процес не використовує у однаковий час спільні з ним структури даних.

Розповсюдженими взаємовиключеними алгоритмами є :

– розподілений;
– централізований алгоритм;
– алгоритм маркерного кільця.

2.11.1 Централізований алгоритм

При спробі зайтив критичну зону координатору відправляється запит з вказаною областю. Якщо ні одного з процесів там немає, то координатор надає дозвіл на використання області. У іншому випадкуприходить відповідь «доступ заборонено», а даний запит стає у чергу на область. Коли роботи завершуються, процес надішле повідомлення про вихід з критичної зони.

Централізований алгоритм надає простішийзасіб організації взаємовиключень в розподілених системах. На рис. 2.4 реалізація централізованого алгоритму для процесів 0, 1, 2. Припустимо, що є інший процес 1, який просить дозвіл на вхід у ту ж область. Координатор знає, що у області вже є процес, а тому не надає дозволу на вхід і ставить запит в чергу та очікуєнаступних повідомлень [2,4].

Коли настає момент, що процес 1 покидає критичну зону, координатору надсилається повідомлення, у якому відмова від доступу на область. Далі

координатор обирає перший елемент з черги відкладених запитів та надсилає процесу повідомлення з дозволом на доступ до області 1.

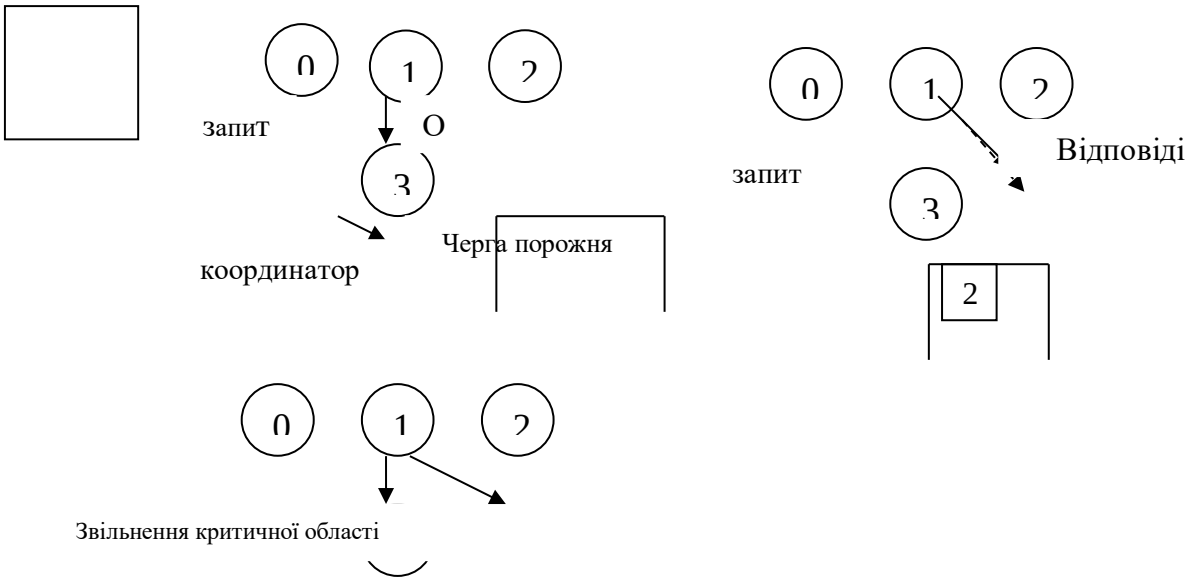


Рисунок 2.4 – Реалізація централізованого алгоритму

Коли процес планує ввійти у критичну область, тоді координатору надсилається повідомлення, у якому повідомляється в яку область він намагається увійти та запитується дозвіл на це. Якщо у даний момент жоден процес не знаходиться в критичній зоні, координатор дає дозвіл на доступ.

Для алгоритму потрібна наявність повної упорядкованості подій у системі. У кожній парі подійповинно бути однозначно зрозуміло, яке з подій відбулося першим.

2.11.2 Розподілений алгоритм

Розподілений алгоритмвводить впорядкованість довільної пари подій. Намагаючись зайтив критичну зону створюється повідомлення з назвою критичної області, власним номером та поточним часом, після чого

відправляється всім процесам. При отриманні іншими процесами запиту, можливі три варіанти подій:

– коли процес намагається увійти в область, але щене ввійшов, то цей процесспівставляє мітки часу;

– коли процес вже в області, то він не відповідає.Номер запиту розміщуєтьсяв чергу;

– отримувач відправляє повідомлення «ОК», коли він не знаходиться та не планує знаходитись в критичній зоні.

На рис. 2.5 2 процеси намагаються ввійти у критичну область. Ці процеси 0, 2 мають конфлікти; процес 0 з меншою відміткою часу, а тому виграє 2 процес. Після цього процес 0 завершує роботу в критичній області і видає повідомлення «ОК», звільняючи область для 2.

2.11.3 Алгоритм маркерного кільця

В цьому алгоритмі програмно реалізується логічне кільце, в якому визначаємо положення кожного з процесів у кільці. За маркером (token) кожен наступний процес перевіряє, чи є можливість для входження в критичну зону, як показано на рисунку 2.6.

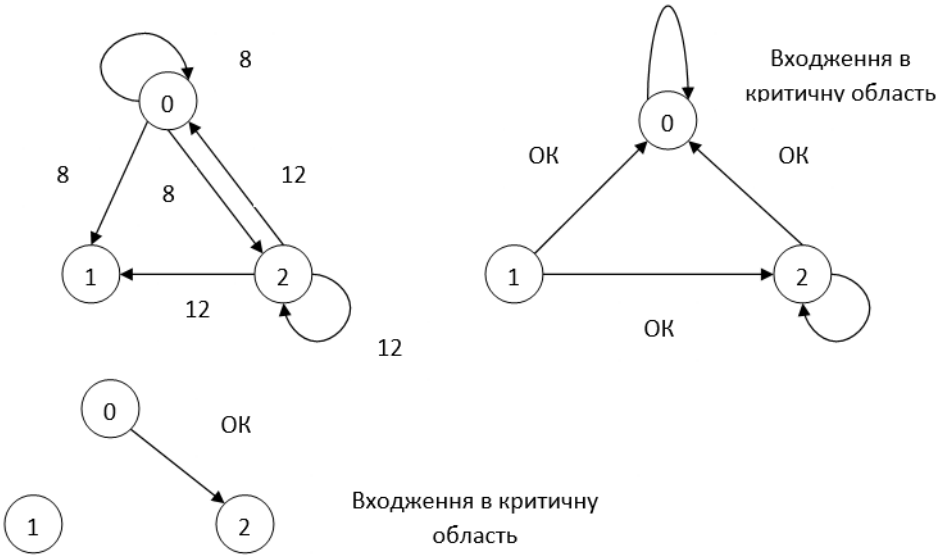


Рисунок 2.5 – Схема спроби входження в критичну область

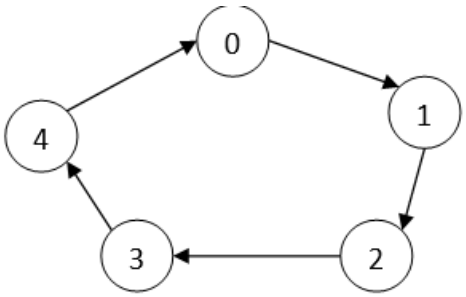


Рисунок 2.6 – Алгоритм маркерного кільця

Якщо увійти у критичну зону, то процес входить, а після виконаної роботи він передає маркер наступному за ним процесу.

Всі процеси становлять логічне кільце, коли кожен знає, хто слідує за ним. По кільцю циркулює маркер, що дає право на вхід в критичну секцію. Отримавши маркер (за допомогою повідомлення) процес або входить в критичну секцію (якщо він чекав дозволу) або переправляє маркер далі.

Після виходу з критичної секції маркер переправляється далі, повторний вхід в секцію при тому ж маркері забороняється.

При реалізації алгоритму виникає проблема. Якщо якийсь процес перестане функціонувати, то алгоритм не працює. Однак відновлення простіше, ніж в інших випадках. Наявність квитанцій дозволить виявити такий процес в момент передачі маркера (якщо поломка сталася поза критичного

інтервалу). Переставши функціонувати процес повинен бути виключений з логічного кільця, для цього доведеться кожному знати поточну конфігурацію кільця

2.11.4 Маркерний деревовидний алгоритм

Усі процеси це збалансовані двійкові дерева. Кожен з процесів має чергу запитів (від себе і сусідніх процесів) та показник власника маркера. Вхід в критичну секцію:

- Якщо є маркер, то процес виконує КС.
- Якщо маркера немає, то процес розміщує запит в чергу запитів та посилає повідомлення ЗАПИТ у напрямку власника маркера. Після чого чекає повідомлень.

Поведінка процесу при прийомі повідомлень

Процес, що не знаходиться всередині КС має реагувати на повідомлення 2 видів - МАРКЕР та ЗАПИТ.

А) Прийшло повідомлення «МАРКЕР»

М1. Взяти перший запит черги та послати маркер автору повідомлення;

М2. Поміняти значення покажчика в бік маркера;

М3. Виключити запит з черги;

М4. Якщо в черзі залишилися запити, то послати повідомлення «ЗАПИТ» в сторону маркера.

Б) Прийшло повідомлення «ЗАПИТ».

Необхідно помістити запит у чергу.

Коли немає маркера, то відіслати повідомлення ЗАПИТ у сторону маркера, інакше - перейти на пункт М1.

Вихід з критичної секції

Якщо черга запитів порожня, то при виході нічого не робиться, інакше - перейти до пункту М1.

2.11.5 Децентралізований алгоритм на основі тимчасових міток

Потрібно глобальне впорядкування всіх подій в системі за часом.

Коли процес хоче ввійти у критичну зону, тоді він посилає всім процесам повідомлення-запит, у якому міститься ім'я критичної зони, номер процесу та поточний час. Одразу після відправки запиту процес чекає, допоки дадуть йому дозвіл. Після отримання від усіх дозволу, він входить в критичну секцію [3, 11].

Коли процес отримує повідомлення-запит, в залежності від свого стану по відношенню до зазначеної критичної секції він діє одним із таких способів. Після виходу з секції він посилає повідомлення «ОК» всім процесам, запити від яких він запам'ятав, а потім стирає всі успішної реєстрації запитів.

На одне проходження секції кількість повідомлень - 2 (n-1), n - число процесів. Окрім того, одна критична точка замінилася на n точок (якщо якийсь процес перестане функціонувати, то відсутність дозволу від нього всіх зупинить).

Деякі поліпшення алгоритму вимагають наявності неподільних широкомовних розсилок повідомлень.

2.11.6 Алгоритм широкомовного маркеру

Маркер містить: чергу запитів; масив LN [1 ... N] з номерами останніх опрацьованих запитів.

Якщо процес Pk, запитує критичну секцію і не містить маркера, тоді збільшується порядковий номер запитів RNk [k] і посилається широкомовне повідомлення «ЗАПИТ», яке містить номер процесу (k) та номер запиту (Sn = RNk [k]). Цей процес Pk виконує критичну зону, якщо має маркер.

Коли процес Pj отримає повідомлення-запит від процесу Pk, він встановлює RNj [k] = max (RNj [k], Sn). Коли Pj має вільний маркер, тоді маркер посилається Pk тільки в випадку, коли RNj [k] == LN [k] +1.

Встановлює LN [k] у маркері рівним RNk [k]. Для кожного Pj, у якого RNk [j] = LN [j] +1, додається ідентифікатор у маркерну чергу запитів.

Коли маркерна черга запитів не порожня, тоді із неї видаляється 1-ий елемент, а маркер надсилається відповідному процесу (запит якого був першим у черзі).

2.11.7 Порівняння алгоритмів

Основні характеристики алгоритмів показано у таблиці 2.1.

Таблиця 2.1 - Характеристики алгоритмів голосування

Алгоритм	Число повідомлень на вхід-вихід	Затримка перед входом в число повідомлень	Проблеми алгоритму
Централізований	3	2	Крах координатора
Розподілений	2(n-1)	2(n-1)	Збій в одному з процесів
Маркерного кільця	Від 1 до ∞	Від 0 до n-1	Втрата маркера, збій в одному з процесів

2.12 Розподілені транзакції

Розподілені транзакції являють собою виконання дій, які обов'язково узгоджені між кількома процесами. У випадку збою чи то відмові транзакція може бути розпочата з того стану системи, що був до її початку.

Транзакції перетворюють процеси доступу і модифікації багатьох елементів даних у одну атомарну операцію. Коли процес під час транзакції приймає рішення зупинитись та повернути назад, то усі дані відновлюються з значеннями, у яких вони були до початку даної транзакції [10, 12].

Властивості розподілених транзакцій (ACID):

1) atomic - атомарність, яка передбачає, що для світу транзакція є неподільною і розглядається як одне ціле;

2) consistent - несуперечність, коли транзакція не порушує інваріанти системи;

3) isolated - ізольованість, яка передбачає для одночасно виконующихся транзакцій відсутність впливу одна на одну;

4) durable - довгостроковість, тобто після завершення транзакції внесені нею зміни - постійні.

Транзакції також можна класифікувати за ознаками взаємної залежності:

1. flat transaction - пласкі транзакції, для яких виконується ACID;

2. nested transaction - вкладені транзакціїї, у яких ієрархічна структура;

3. distributed transaction - розподілені транзакції, в яких передбачається виконання окремих частин транзакції на різних машинах.

Транзакція - це серія операцій, задовольняюча властивості ACID. Пласкі транзакції найбільш прості та часто використовувані, але мають помітні обмеження. До яких можна віднести неспроможність дати частковий результат, коли є завершення / переривання транзакції.

Вкладені транзакції – транзакції, які логічно розділяються на ієрархічно-організовані підтранзакції [2]. Розподілені транзакції також включають випадки, коли вкладені (пласкі) транзакції працюють с даними, розподіленими на декілька машин. А розподілені транзакції виступають як логічно пласкі, неподільні транзакції, працюючі з розподіленими даними.

Основна проблема розподілених транзакцій заключається у тому, що для підтвердження транзакції та блокування даних потрібні спеціальні розподілені алгоритми. На рисунку 2.7 можна побачити відмінності розподіленних і вкладених транзакцій.

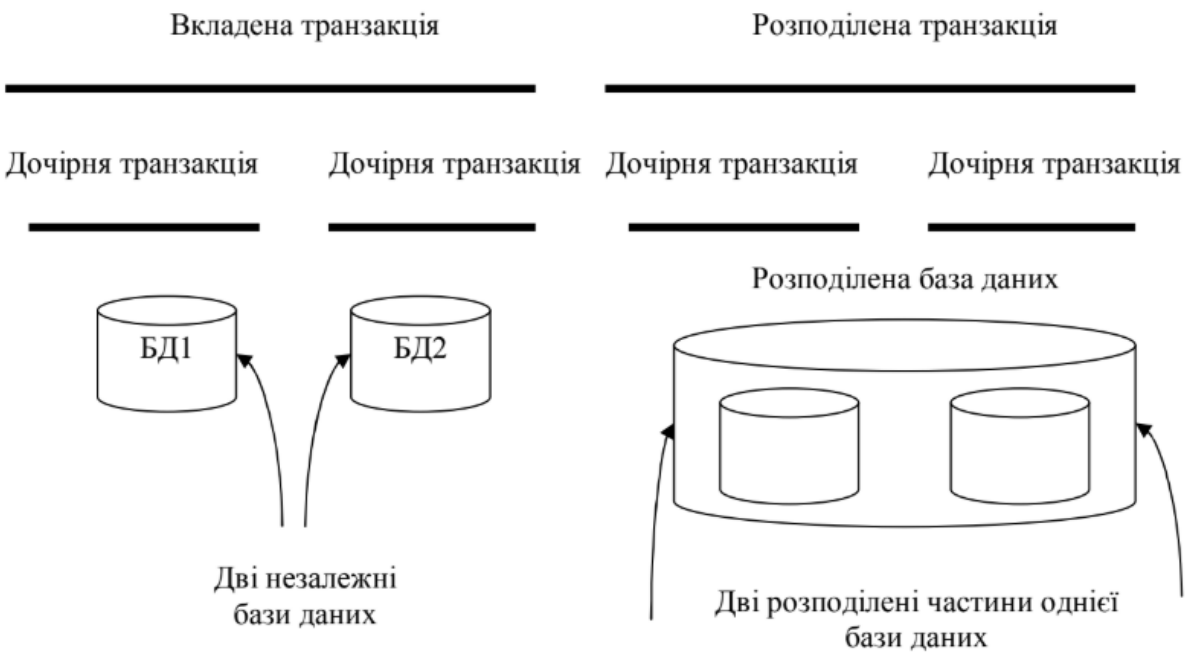


Рисунок 2.7 - Схема відмінностей розподіленних та вкладених транзакцій

У файлових системах для реалізації транзакцій є два методи: журнал з упереджуючим записом (write-ahead log) та журнал закритого робочого простору.

Закритий робочий простір - це набір файлів, до яких транзакція у момент реалізації намагається отримати доступ. Але проблема оптимізації робочого

простору при зчитуванні незмінуванного файла полягає в створенні закритої зони, яка містять лише вказівник на робочий простір свого родича.

При відкритті файла на зчитування, процес йду за вказівником, доки не знайде в робочому просторі батька або більш віддаленого предка [3, 4].

При відкриванні на запис файл шукається аналогічно зчитуванню, тільки спочатку копіюється в закритий робочий простір. При модифікації блоку в робочому просторі створюється новий індекс із посиланням на копії модифікованих блоків (див. рисунок 2.8).

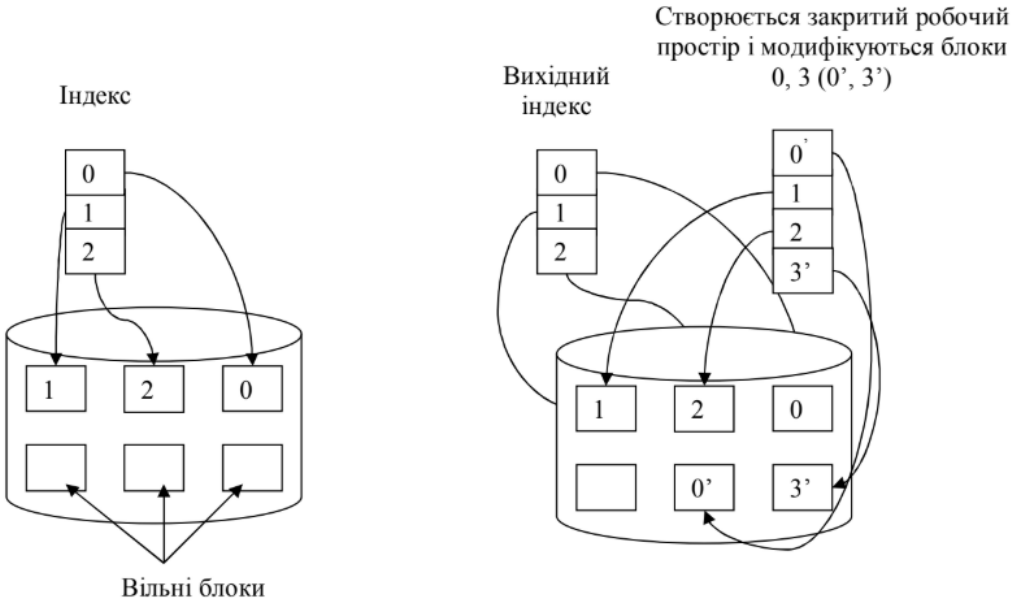


Рисунок 2.8 – Схеми модифікації блоків у робочому просторі

Після завершення транзакції закритий робочий простір видаляється, а закриті блоки стають вільними для доступу. Модифіковані блоки (0', 3') переміщуються у робочий простір предка.

Журнал з упередженим записом включає фіксацію змін у файлі. Після запису у журнал зміни заносяться до файла. При перериванні транзакції файл відновлюється за даними журнала.

Для підвищення продуктивності розподілених систем існує паралельне виконання транзакцій з допомогою засобів керування. До таких засобів належать:

- менеджер даних, який здійснює операції читання та запису даних на фізичному рівні;
- планувальник, який відповідає зп управління паралельною роботою транзакцій;
- менеджер транзакцій, що відповідає за атомарність та довговічність транзакцій. Також оброблює примітиви транзакцій, перетворючи їх на запит до планувальника.

Паралельні транзакції виконуються одночасно над одним набором даних спільного використовування. Метою керування є дозвіл кільком транзакціям виконуватися одночасно, але так, щоб набір оброблювальних елементів даних залишався несуперечливим. Несуперечність досягається через доступ транзакцій к елементам даних у певному порядку так, щоб кінцевий результат був як при виконанні транзакцій послідовно.

Головна задача планувальника у правильному плануванні паралельним виконанням конфліктуючих операцій. Якщо дві операції конфліктують, то вони працюють з тими ж елементами даних.

Поширені конфлікти читання-запису та подвійного запису, незалежно від того, належать вони одній або різним транзакціям. Алгоритм керування паралельними виконаннями транзакцій класифікують за способом синхронізації операцій читання та запису. Синхронізація може виконуватись за механізмом явного упорядкування операцій з допомогою відміток часу або взаємного виключення загально використовуваних даних.

Широко відомими є алгоритми двохфазного блокування із песимістичним або оптимістичним за відмітками часу. У алгоритмі двохфазного блокування 2PL процес під час транзакції звертається до планувальника з запитом блокування даних. Планувальник на фазі підйому визначає всі необхідні блокування, а потім, на фазі спаду знімає їх.

Модифікація алгоритму - режим строгого двохфазного блокування. За цим алгоритмом фаза спадання починається після завершення спільно

використовуваних транзакцій підтвердження / керування. При двохфазному блокуванні можливі тупіки.

Технологія їх уникнення - блокування у заданій послідовності або використання тайм-ауту, який перевищує час блокування.

Відомі кілька способів реалізації двохфазного блокування:

1) первинне двохфазне блокування;

2) централізоване двохфазне блокування;

3) розподілене двохфазне блокування.

Висновки до розділу 2

В даному розділі було розглянуто відомі алгоритми синхронізації процесів. Визначено основні засоби для вирішення проблем синхронізації процесів в розподілених системах для керування процесами в одному адресному просторі та в різних адресних протосторах виконання процесів.

Визначено основні переваги та недоліки механізмів синхронізації. Детально розглянуто елпмтіфікацію і принципи приниципи роботи розподілених транзакцій. Визначено роль менеджера транзакцій, планувальника транзакцій та менеджера даних.

РОЗДІЛ 3 РЕАЛІЗАЦІЯ СИСТЕМИ РОЗПОДІЛЕНИХ ТРАНЗАКЦІЙ

3.1 Алгоритмічні рішення задач взаємодії процесів

3.1.1 Опис взаємодії процесів розподіленої системи

У процеса є адресний простір, його стан характеризується наступним:

- замовлення на введення-виведення;
- дескриптори файлів;
- таблиці сторінок / сегментів;
- регістри;

Великий обсяг цієї інформації робить доцільним застосування легких процесах і потоків (threads), застосування яких розпочалось ще для однопроцесорних ЕОМ (фізичні процеси або їх моделювання, поєднання обмінів даними). Потоки просто необхідні при застосуванні переваг багатопроцесорних ЕОМ з загальною пам'яттю.

Процеси можуть бути незалежними та не мати потреби в будь-якій синхронізації і обміні інформацією (але можуть конкурувати за ресурси), або можуть бути взаємодіючими.

Взаємодія процесів за умови реалізації для багатьох процесів (або ниток) відбувається двома основними способами:

- за допомогою поділу пам'яті (оперативної або зовнішньої)
- за допомогою передачі повідомлень

При взаємодії через загальну пам'ять процеси повинні синхронізувати своє виконання. Розрізняють два види синхронізації - взаємне виключення критичних інтервалів і координацію процесів.

Взаємодія у критичних областях призводить до недермінізму виконання операцій, що призводить до перегонів.

Результат залежить від порядку виконання цих команд. Потрібно взаємне виключення критичних інтервалів.

Рішення проблеми взаємного виключення повинно задовольняти вимогам:

– процес не повинен нескінченно довго чекати дозволу на вхід у критичний інтервал;

– якщо жодного процесу немає у критичному інтервалі, то будь-який процес, бажаючий увійти, повинен отримати дозвіл без затримки;

– в будь-який час тільки один процес може бути всередині інтервалу;

– не повинно існувати припущень про швидкості процесорів.

3.1.2 Алгоритм Деккера

Блокування зовнішніх переривань, яке може порушуватися управління зовнішніми пристроями, спричиняє можливі внутрішні переривання при роботі з віртуальною пам'яттю.

Взаємне виключення критичних інтервалів у випадку багатопроцесорної ЕОМ потребує відповідних рішень. Зокрема. прикладом програмних рішень є реалізація на основі неподільності операцій запису і читання з пам'яті.

Прикладом такої ефективної реалізації є алгоритм Деккера. Алгоритм Деккера має такий вигляд.

```
int trn;
boo  flg[];

proc( int k )
{
while (TRUE)
```

```
{
<обчислення>;
in_reg( k );
<критичний інтервал>;
out_reg( k );
}
}

void in_reg( intk )
{
try:  flg[ k ]=TRUE; while (flag [( k+1 ) % 2])
{
if ( trn = = k ) continue;
flg[ k ] = FALSE;
while ( trn != k );
goto try;
}
}

void out_reg( int k )
{
trn = ( k +1 )  % 2;
flag[ i ] = FALSE;
}
trn = 0;
flg[ 0 ] = FALSE;
flg[ 1 ] = FALSE;
proc( 0 ) AND proc( 1 )  /* запустили 2 процеса */
```

Іншим прикладом є реалізації алгоритм у Петерсона.

```
int trn;
int flg [];

void in_reg (int k)
```

```
{
int othr; / * Номер іншого процесу * /

othr = 1 - k;
flag [k] = TRUE;
turn = k;
while (trn = = k&& flg [othr] = = TRUE) / * порожній оператор * /;
}

void out_reg (int k)
{
flg [k] = FALSE;
}
```

3.1.3 Використання неподільних операцій

Неподільної операції передбачають виконання операції з початку до кінця без наявних помилок. Таке виконання є подібним до транзакції і використовуються для роботи з критичними областями.

Використання неподільної операції можна продемонструвати на прикладі команд TSL. Операція має формат:

$$TSL\ (r, s): [r = s; s = 1]$$

Квадратні дужки - використовуються для специфікації неподільності операцій.

Програмна реалізація на Асемблер може мати такий вигляд.

```
enter_region:
tsl reg, flag
```

```
cmp reg, # 0 / * порівнюємо з нулем * /
jnz enter_region / * якщо не нуль - цикл очікування * /
ret
leave_region:
mov flag, # 0 / * присвоюємо нуль * /
ret
```

3.1.4 Реалізація семафорів і їх застосування

Семафори Дейкстри є невід'ємною цілою змінноюа, що може перевірятись та змінюватись за допомогою тільки двох функцій P (s) та V (s).

Функція запиту семафора P (s) має такий вигляд.

[If (s == 0) <заблокувати поточний процес>; else s = s-1;]

Функція звільнення семафора V (s) має такий вигляд.

[If (s == 0) <розблокувати один із заблокованих процесів>; s = s + 1;]

Семафори часто використовують для реалізації проблем синхронізації процесів і потокі. Поширеним є застосування для взаємного виключення критичних інтервалів і для координації у проблемі виробник-споживач.

Алгоритм реалізації процесу для програми виробник-споживач для двох процесів (відомий, як проблема для обмеженого буфера) на основі семафорів має такий вигляд.

```
semaphore  s = 1;
semaphore  full = 0;
semaphore  empty = N;
```

```
prod()      ¦   cons()
{           ¦   {
¦
int itm;        ¦       int itm;
while (TRUE)    ¦       while (TRUE)
{           ¦       {
prod_itm(&itm);     ¦
P(emp);     ¦           P(fll);
P(s);       ¦           P(s);
in_itm(itm);    ¦           out_itm(&itm);
V(s);       ¦           V(s);
V(fll);     ¦           V(emp);
¦           cons_itm(itm);
}           ¦       }
}           ¦   }
¦
prod() AND cons()  /* запустили 2 процеса */
```

Дане рішення може застосовуватись і для реалізації процесів у різних адресних просторах.

3.1.5 Використання семафорів для мультипрограмного режиму

Існують різні способи використання семафорів для мультипрограмного режиму. Поширеними є:

– блокування зовнішніх переривань;
– заборона перемикання на інші процеси;
– змінна і черги чекають процесів в ОС.

Для багатопроцесорної ЕОМ перші два способи не годяться. Для реалізації третього способу досить команди TSL і можливості оголошувати переривання вказаною процесору.

Блокування процесу і перемикання на інший - не ефективно, якщо семафор захоплюється на короткий час. Час очікування звільнення семафорів може бути реалізовано за допомогою циклічного опитування значення семафора. Недоліки такого "активного очікування" - марна трата часу, навантаження на загальну пам'ять, і можливість фактично заблокувати роботу процесу, що знаходиться в критичному інтервалі. Тобто якщо даний об'єкт використовується багатьма іншими, то семафори не годяться.

3.1.6 Відслідковування подій

Події подають змінними, що показують, що відбулися певні події. Для оголошення події служить оператор POST (ім'я змінної), для очікування події - WAIT (ім'я змінної). Для очищення (привласнення нульового значення) застосвують оператор CLEAR (ім'я змінної).

Варіанти реалізації передбачають можливість не зберігати інформацію. Так по по оператору POST з очікування виводяться тільки ті процеси, які вже виконали WAIT. Можливо одноразове оголошування, за яким немає оператора очистки.

Метод послідовної верхньої релаксації (SOR) з використанням масиву подій дозволяє обробляти масиви подій. Приклад реалізації має такий вигляд.

```
float A [L1] [L2];
struct event s [L1] [L2];
for (i = 0; i <L1; i ++)
for (j = 0; j <L2; j ++) {clear (s [i] [j])};
for (j = 0; j <L2; j ++) {post (s [0] [j])};
```

```
for (i = 0; i <L1; i ++) {post (s [i] [0])};
.....
.....
parfor (i = 1; i <L1-1; i ++)
parfor (j = 1; j <L2-1; j ++)
{Wait (s [i-1] [j]);
wait (s [i] [j-1]);
A [i] [j] = (A [i-1] [j] + A [i + 1] [j] + A [i] [j-1] + A [i] [j + 1]) / 4;
post (s [i] [j]);
}
```

3.1.7 Синхронізація процесів на основі обміну повідомленнями

Обмін повідомленнями (message passing) використовує примітиви мікроядра операційної системи. Поширеними є застосування примітивів:

```
snd (dst, & msg, msz);
rcv ([src], & msg, msz);
```

Команди дозволяють створити віртуальний буферний простір елементів даних для реалаізції різних модлей (проблем) синхронізації розподілених процесів.

Зокрема, реалізація може передбачати буферизацію (поштові скриньки). Реалізація пайпів як поштових скриньок замінюють файли і не зберігають повідомлення.

Приклад використання буферизованих повідомлень для моделі виробники-споживачі має такий вигляд.

```
#define M 1000 / * максимальне число повідомлень * /
/ * В буфері
```

```
#define msz 5 / * розмір повідомлення * /
typedef int msg [msz];

prod ()
{
msg m;
int itm;

while (TRUE)
{
prod_itm (& itm);
rcv (cons, & m, msz); / * Отримує порожній * /
/ * "Контейнер" * /
bld_msg (& m, itm); / * Формує повідомлення * /
snd (cons, & m, msz);
}
}

cons ()
{
msg m;
int itm, i;

for (i = 0; i <M; i ++)
snd (prod, & m, msz); / * Посилає всі порожні *.
/ * "Контейнери" * /
while (TRUE)
{
rcv (prod, & m, msz);
extr_itm (& m, itm);
snd (prod, & m, msz); / * Повертає "контейнер" * /
cons_itm (itm);
}
}
prod () AND cons () / * запустили 2 процеси * /
```

Механізми семафорів і обміну повідомленнями взаємозамінні семантично і на мультипроцесорах можуть бути реалізовані один через інший.

3.1.8 Планування процесорів

Планування процесорів суттєво впливає на продуктивність мультипроцесорної системи. Можна виділити такі основні чинники деградації продуктивності.

Витрати на перемикання процесора можна визначити не тільки як перемикання контекстів, а й переміщеннями сторінок віртуальної пам'яті, а також псуванням кеша. Інформація в кеші іншому додатку не потрібна і буде замінена.

Перемкання на другий процес у момент, коли поточний процес виконує критичну секцію, а інші процеси активно очікують входу в секцію. У цьому випадку втрати будуть великі (хоча ймовірність переривання виконання коротких критичних секцій мала).

Для боротьби з деградацією продуктивності застосовують наступні стратегії:

– процеси плануються на ті процесори, на яких вони виконувалися в момент їх зняття.

– планування, коли процеси знаходяться в критичній секції і не перериваються, а активно очікують входу в критичну секцію.

– спільне планування, при якому всі процеси програми одночасно знімаються з них для скорочення перемикань контексту;

– планування з урахуванням "рад" програми. В ОС Mach є два класи таких рад (hints) - вказівки (різного ступеня категоричності) про зняття з

поточною діяльністю з процесора та вказівки про процес, який повинен бути замість поточного.

3.1.9 Опис розподіленої системи

У даній роботі за основу вибрано приклад простої розподіленої системи як певного підприємства з філіалами в різних містах. Кожен філіал має свій сервер на якому знаходиться база даних з інформацією, якою оперує даний філіал. Припустимо, що ми маємо головне відділення, яке оперує даними з усіх філіалів, як показано на рис. 3.1.

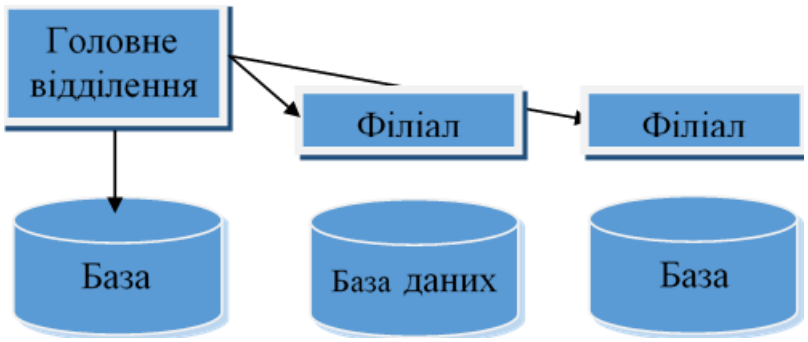


Рисунок 3.1 – Схема розподіленої системи

Поставлена задача – розробити механізми взаємодії з даними та синхронізувати процеси в системі для усунення суперечливостей.

Для побудови даної системи було використано систему керування базами даних MySQL та середовище .NETCore. В ролі менеджера та планувальника транзакцій виступають WebAPI, що спілкуються між собою за допомогою протоколу http. В ролі менеджера даних – БД.

3.1.10 Системи розподілених транзакцій

Розберемо те, як функціонують елементи. Менеджер транзакцій (що в описаному випадку буде розгорнутий на сервері головного відділення) чекає на вхід JSON, що містить в собі набір команд, що власне й являють собою транзакцію. Після отримання списку команд менеджер транзакцій окремо обробляє кожну з них, відсилаючи запити на виконання відповідних дій в заявлені в команді відділення. Далі на серверах відділень працює по одному планувальнику задач, кожен з яких чекає запиту. Після отримання запиту на читання повертаються дані з попереднього збереженого стану системи (до будь-яких змі). А при отриманні запиту на запис вибудовується черга в порядку надходження запитів, і по черзі виконується запис (при цьому самі дані блокуються) (Додатки А-Е).

При виконанні збереження, перевіряється черга на запис – якщо пуста, то зберігати, якщо ні – почекати поки не обробиться черга до того елементу, який був останнім в момент запуску процедури збереженняв включно. Операція припинення працює аналогічно, крім того що не чекає звільнення черги, а банально чистить її до того ж елементу. На рис. 3.2. видно макет кінцевої системи.

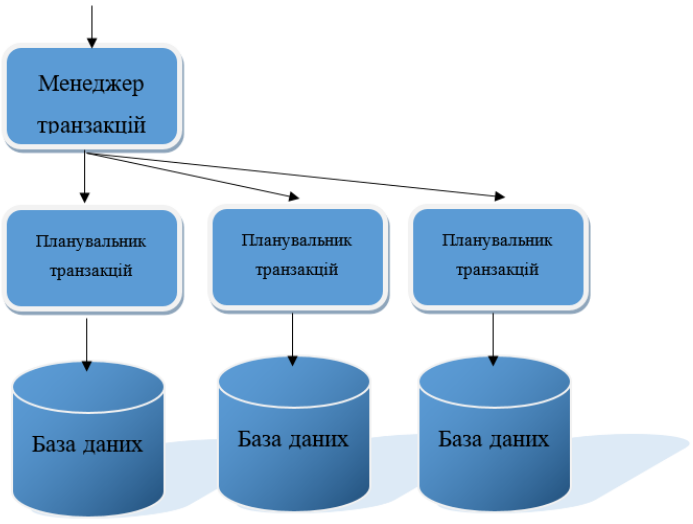


Рисунок 3.2 – Схема прикладу системи розподілених транзакцій

3.2 Клієнтський додаток для роботи з системою

3.2.1 Головне вікно програми

Для того щоб спростити роботу користувача з системою, тобто позбавити його необхідності вручну надсилати запити до планувальника, а також для зручного перегляду даних (наявність фільтрації, наочна візуалізація) було розроблено додаток для персональних ком'ютерів з операційною системою Windiws. Засоби, що використані при розробці: мова програмування - C#, система для побудови клієнтський додатків - Windows Presentation Foundation (WPF). На рис 3.3 видно кінцевий результат.

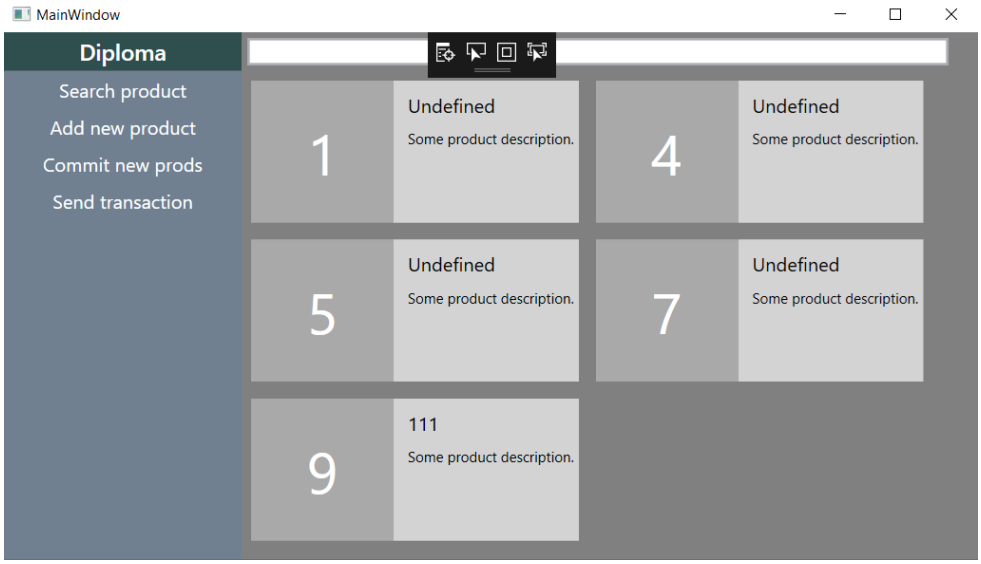


Рисунок 3.3 – Головне вікно програми (вкладка читання даних)

Як видно з рис 3.3 додаток має панель навігації зліва, що дозволяє нам подорожувати по вкладках та надіслати сформовану транзакцію.

3.2.2 Головне вікно програми

При запуску додатку першою запалюється вкладка перегляду даних, на ній дані показані у вигляді сітки карток, на яких видно короткий опис об’єкта, його ім’я та унікальний ідентифікатор.

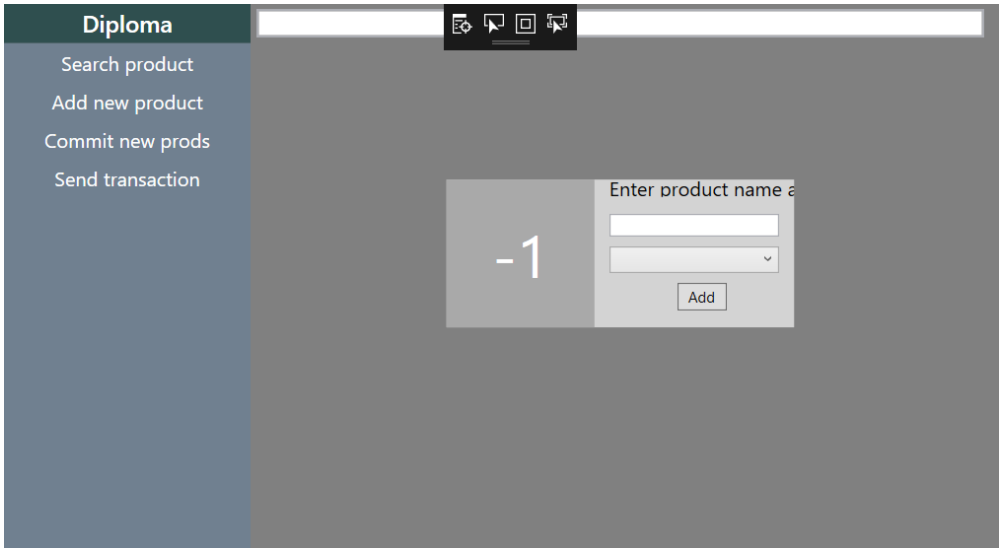


Рисунок 3.4 – Вкладка додавання об’єкту

Наступною ми маємо вкладку додрвання об’єкту (рис. 3.4). На ній користувач має змогу додати продукт до свого запиту. З введених даних формується команда запису й додається до списку всіх команд, які були сформовані до цього.

3.2.3 Інтерфейс збереження/скасування транзакції

На вкладці збереження/скасування тразакцій (Рис 3.5) відображаються всі об’єкти які були додані в запит, але не були ні відбравлені на обробку, ні збережені в рамках черги команд, що має обробити менеджер транзакцій.

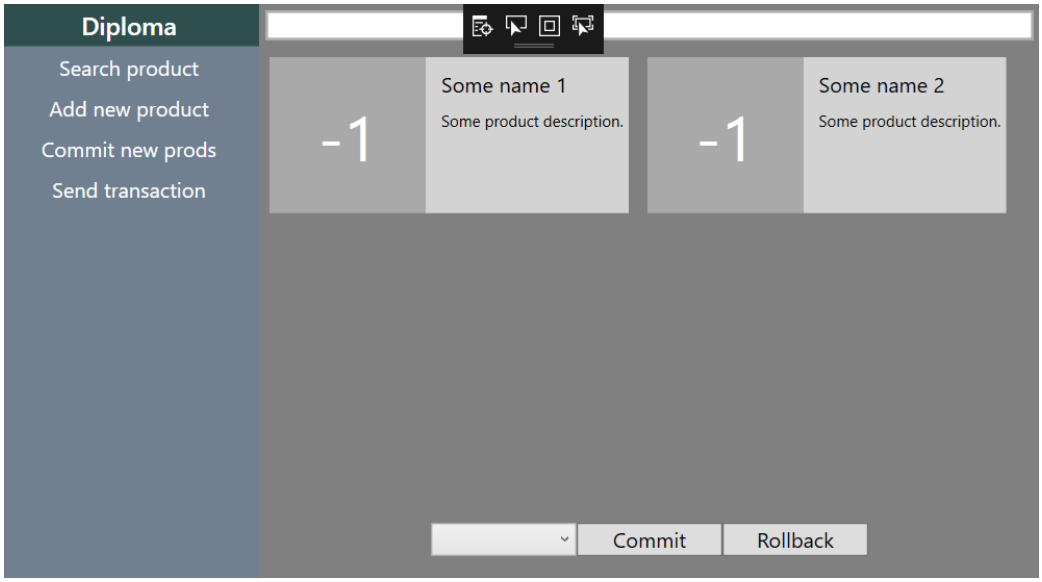


Рисунок 3.5 – Вкладка збереження/скасування транзакції

За допомогою кнопок, що видно внизу рисунка 3.5, можна додати команду збереження та скасування змін до списку команд.

Висновки до розділу 3

Проведено дослідження вимог до інтерфейсу при керування транзакціями розподілених систем у системах з планувальниками транзакцій.

Описано процес розробки прикладу системи розподілених транзакцій та клієнтського додатку для роботи з ним.

Розроблено програмне забезпечення інтерфейсу розподіленої системи на основі розподіленихз транзакцій для розподілених баз даних.

Розроблено програмне забезпечення виконання транзакцій у розподілених системах для різних режимів керування розподіленими процесами.

4 ФУНКЦІОНАЛЬНО-ВАРТІСНИЙ АНАЛІЗ ПРОГРАМНОГО ПРОДУКТУ

4.1 Постановка задачі техніко-економічного аналізу

У даному розділі проводиться оцінка основних характеристик програмного забезпечення, а саме системи розподілених транзауцій та клієнтського додатку для роботи з нею. Інтерфейс користувача створений за допомогою технологій WPF.

Програмний продукт розроблений у вигляді двох Web API та додатку для настільних комп’ютерів з ОС Windows.

У роботі застосовується метод ФВА для проведення техніко-економічного аналізу розробки.

Відповідно до цього варто обирати і систему показників якості програмного продукту.

Технічні вимоги до продукту наступні:

- програмний продукт має мати зручний користувацький інтерфейс;
- система розподілених транзакцій має забезпечувати несуперечливість даних;
- продукт повинен бути розроблений для підтримки великої кількості користувачів;
- забезпечувати високу швидкість обробки великих об’ємів даних у реальному часі;
- забезпечувати зручність і простоту взаємодії з користувачем або з розробником програмного забезпечення у випадку використовування його як модуля;

– передбачати мінімальні витрати на впровадження програмного продукту

4.1.1 Обґрунтування функцій програмного продукту

Головна функція F_0 – розробка програмного продукту, здатного у зручній для користувача формі отримати на вхід аудіо, і повернути найімовірніші класи звуку. Виходячи з конкретної мети, можна виділити наступні основні функції ПП:

F_1 – вибір оптимального засобу для розробки Web API;

F_2 – вибір засобу для розробки клієнтського додатку;

F_3 –платформа, на яку продукт буде орієнтовано.

Кожна з основних функцій може мати декілька варіантів реалізації.

Функція F_1:

а) платформа .NetCore (мова програмування C#);

б) фреймворкDjango (мова програмування Python).

Функція F_2:

а) засіб для створення клієнтських додатків Windows PresentationFoundation (мова програмування C#);

б) засіб для створення клієнтських додатків Qt (мова програмування C++);

Функція F_3:

а) ОС Windows;

б) ОС Linux.

4.1.2 Варіанти реалізації основних функцій

Варіанти реалізації основних функцій наведені у морфологічній карті системи (Рисунок 4.1). На основі цієї карти побудовано позитивно-негативну матрицю варіантів основних функцій (таблиця 4.1).

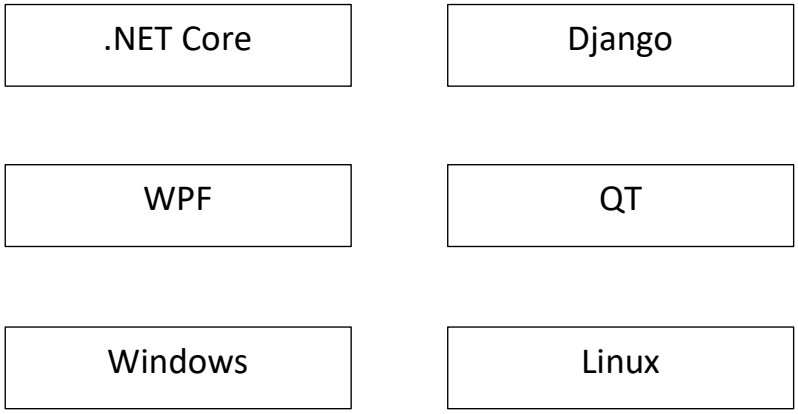


Рисунок 4.1 – Морфологічна карта

Морфологічна карта відображує всі можливі комбінації варіантів реалізації функцій, які складають повну множину варіантів ПП.

Таблиця 4.1 – Позитивно-негативна матриця

Основні функції	**Варіанти реалізації**	**Переваги**	**Недоліки**
F1	*А*	Швидкодія, надійність	Більш висока складність розробки
	Б	Простота в розробці	Менша швидкість роботи
F2	А	Простота, гарна інтеграція зі своєю платформою	Працює лише під ОС Windows
	Б	Кросплатформенність	Більш складний процес розробки
F3	*А*	Популярна платформа серед користувачів	Платна ліцензія
	Б	Безкоштовна ліцензція	Непопулярна платформа серед користувачів

На основі аналізу позитивно-негативної матриці робимо висновок, що при розробці програмного продукту деякі варіанти реалізації функцій варто відкинути, тому, що вони не відповідають поставленим перед програмним продуктом задачам. Ці варіанти відзначені у морфологічній карті.

Функція F1:

Оскільки для даного продукту важливим критерієм є швидкодія та надійність, відкидаємо варіант б).

Функція F2:

Оскільки обидва варіанти можуть бути використані для розробки і пропонують різні переваги, то слід розглянути обидва варіанти

Функція F3:

Оскільки поширеність важливіше, то відкинемо варіант б).

Таким чином, будемо розглядати такий варіант реалізації ПП:

1. F1а – F2а – F3а
2. F1а – F2б – F3а

Для оцінювання якості розглянутих функцій обрана система параметрів, описана нижче.

4.2 Обґрунтування системи параметрів ПП

4.2.1 Опис параметрів

На підставі даних про основні функції, що повинен реалізувати програмний продукт, вимог до нього, визначаються основні параметри виробу, що будуть використані для розрахунку коефіцієнта технічного рівня.

Для того, щоб охарактеризувати програмний продукт, будемо використовувати наступні параметри:

- X1 – швидкодія платформи;
- X2 – частка ринку платформи;
- X3 – кросплатформенність;
- X4 – потенційний об’єм програмного коду.

X1: Відображає швидкодію операцій залежно від обраної мови програмування.

X2: Відображає відносну кількість користувачів окремої платформи.

X3: Відображає кількість підтримуваних платформ.

X4: Показує розмір програмного коду, який необхідно створити розробнику.

4.2.2 Кількісна оцінка параметрів

Гірші, середні і кращі значення параметрів вибираються на основі вимог замовника й умов, що характеризують експлуатацію програмного продукту як показано у табл. 4.2.

Таблиця 4.2 – Основні параметри ПП.

Назва Параметра	**Умовні позначення**	**Одиниці виміру**	**Значення параметра**		
			гірші	**середні**	**кращі**
Швидкодія платформи	X1	Оп/мс	10 000	7 000	4 000
Частка ринку мобільної платформи	X2	%	32	16	8
Кросплатформенність	X3	%	25%	50%	100%

Потенційний об’єм програмного коду	X4	кількість рядків коду	2 000	1 500	1 000

За даними таблиці 4.2 будуються графічні характеристики параметрів рис. 4.2 – 4.5.

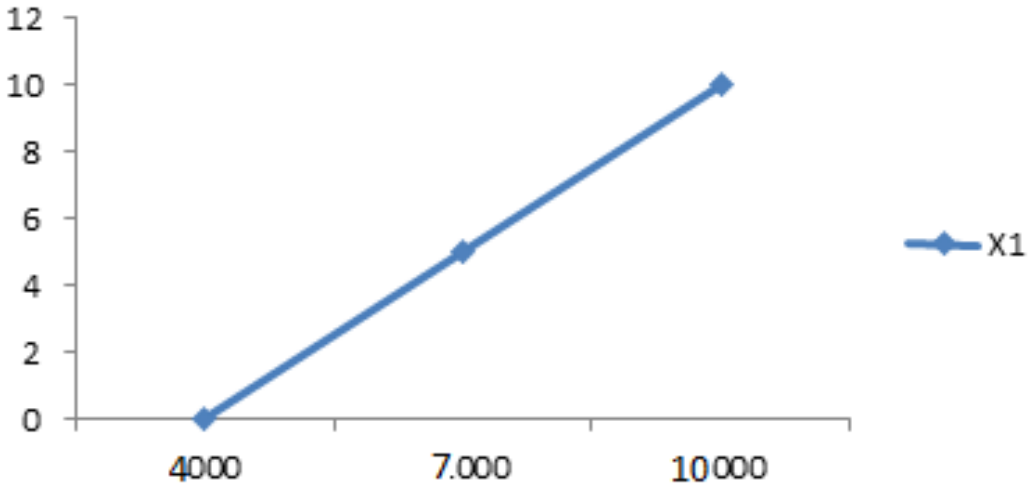


Рисунок 4.2 – X1, швидкодія платформи

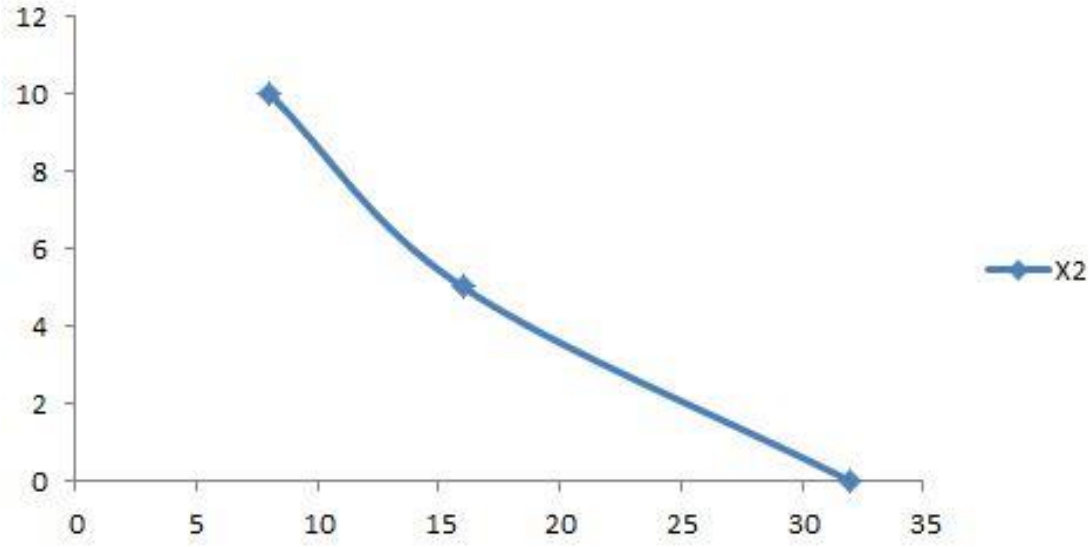


Рисунок 4.3 – X2, Частка ринку платформи

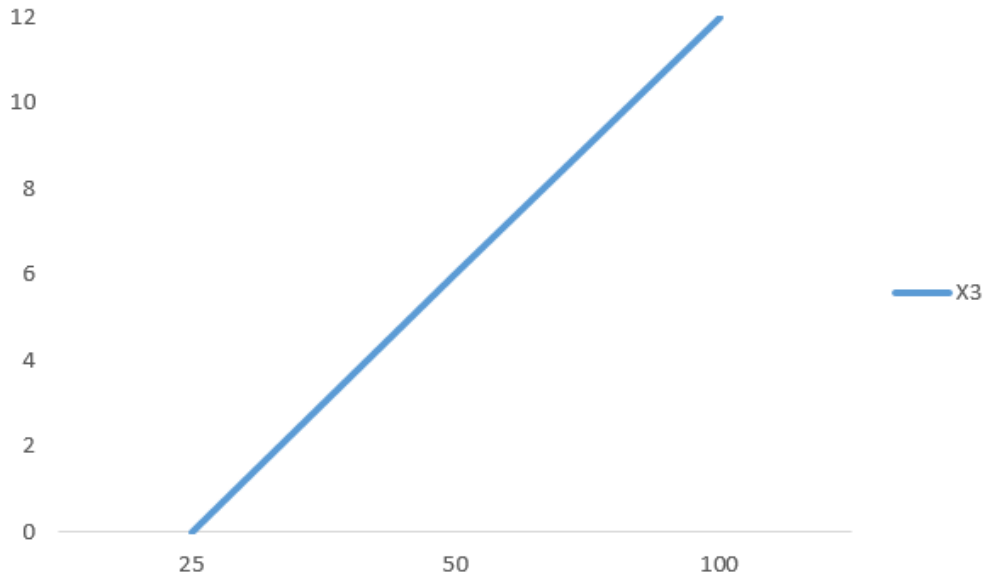


Рисунок 4.4 – X3, кросплатформенність

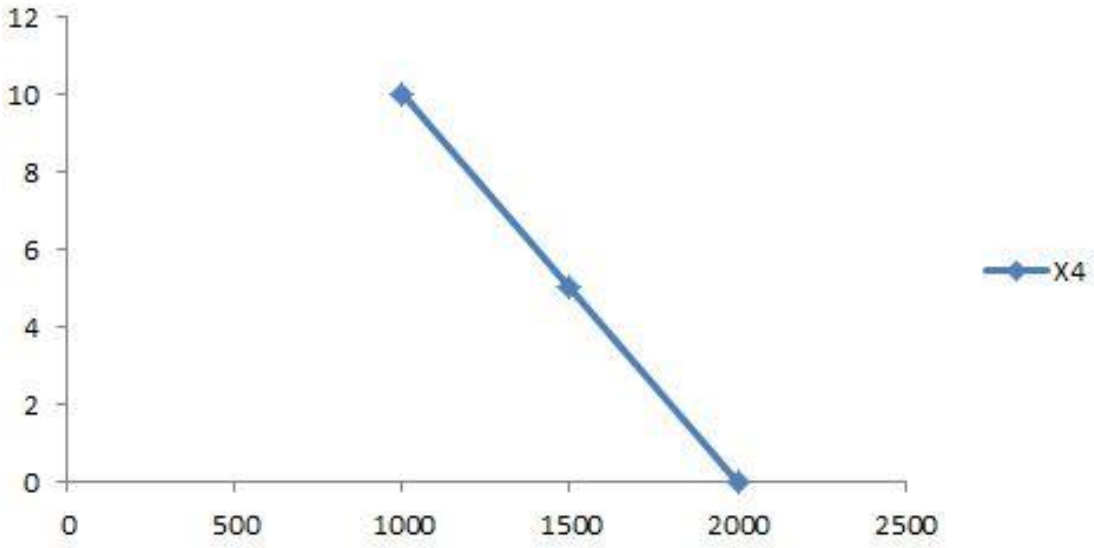


Рисунок 4.5 – X4, потенційний об’єм програмного коду

4.3 Аналіз експертного оцінювання параметрів

Після детального обговорення й аналізу кожний експерт оцінює ступінь важливості кожного параметру для конкретно поставленої цілі – розробка програмного продукту, який дає найбільш точні результати при знаходженні параметрів моделей адаптивного прогнозування і обчислення прогнозних значень.

Значимість кожного параметра визначається методом попарного порівняння. Оцінку проводить експертна комісія із 7 людей. Визначення коефіцієнтів значимості передбачає:

- визначення рівня значимості параметра шляхом присвоєння різних рангів;
- перевірку придатності експертних оцінок для подальшого використання;
- визначення оцінки попарного пріоритету параметрів;
- обробку результатів та визначення коефіцієнту значимості.

Результати експертного ранжування наведені у таблиці 4.3.

Таблиця 4.3 – Результати ранжування параметрів.

Познач. параметра	Назва параметра	Одиниці виміру	Ранг параметра за оцінкою експерта							Сума рангів R_i	Відхи-лення Δ_i	Δ_i^2
			1	2	3	4	5	6	7			
X1	Швидкодія платформи	Оп/мс	1	2	3	1	2	2	1	12	-5,5	30,25
X2	Кількість користувачів платформи	%	3	4	4	3	4	3	4	25	7,5	56,25
X3	Кросплатформенність	%	4	3	2	4	3	4	3	23	5,5	30,25
X4	Потенційний об’єм коду	к-сть рядків коду	2	1	1	2	1	1	2	10	-7,5	56,25
	Разом		10	10	10	10	10	10	10	70	0	173

Для перевірки степені достовірності експертних оцінок, визначимо наступні параметри:

а) сума рангів кожного з параметрів і загальна сума рангів:

$$R_i = \sum_{j=1}^{N} r_{ij} R_{ij} = \frac{Nn(n+1)}{2} = 70,$$

де N – число експертів;

n – кількість параметрів.

б) середня сума рангів:

$$T = \frac{1}{n} R_{ij} = 17{,}5.$$

в) відхилення суми рангів кожного параметра від середньої суми рангів:

$$\Delta_i = R_i - T$$

Сума відхилень по всім параметрам повинна дорівнювати 0;

г)загальна сума квадратів відхилення:

$$S = \sum_{i=1}^{N} \Delta_i^2 = 173.$$

Порахуємо коефіцієнт узгодженості:

$$W = \frac{12S}{N^2(n^3 - n)} = \frac{12 \cdot 173}{7^2(4^3 - 4)} = 0{,}706 > W_k = 0{,}67$$

Рангування можна вважати достовірним, тому що знайдений коефіцієнт узгодженості перевищує нормативний, який дорівнює 0,67.

Скориставшись результатами рангування, проведемо попарне порівняння всіх параметрів і результати занесемо у таблицю 4.4.

Таблиця 4.4 – Попарне порівняння параметрів.

Параметри	Експерти							Кінцева оцінка	Числове значення
	1	2	3	4	5	6	7		
X1 і X2	<	<	<	<	<	<	<	<	0.5
X1 і X3	<	<	>	<	<	<	<	<	0.5
X1 і X4	<	>	>	<	>	>	<	>	1.5
X2 і X3	<	>	>	<	>	<	>	>	1,5
X2 і X4	>	>	>	>	>	>	>	>	1.5
X3 і X4	>	>	>	>	>	>	>	>	1.5

Числове значення, що визначає ступінь переваги і–го параметра над j–тим, аіj визначається по формулі:

$$a_{ij} = \begin{cases} 1.5 \text{ при } X_i > X_j \\ 1.0 \text{ при } X_i = X_j \\ 0.5 \text{ при } X_i < X_j \end{cases}$$

З отриманих числових оцінок переваги складемо матрицю $A = \| a_{ij} \|$.

Для кожного параметра зробимо розрахунок вагомості $K_{вi}$ за наступними формулами:

$$K_{\text{вi}} = \frac{b_i}{\sum_{i=1}^{n} b_i};\ b_i = \sum_{j=1}^{N} a_{ij}.$$

На другому і наступних кроках відносні оцінки розраховуються за наступними формулами:

$$K_{\text{вi}} = \frac{b_i'}{\sum_{i=1}^{n} b_i'},\text{де } b_i' = \sum_{j=1}^{N} a_{ij} b_j.$$

Як видно з таблиці 4.5, різниця значень коефіцієнтів вагомості не перевищує 2%, тому більшої кількості ітерацій не потрібно.

Таблиця 4.5 – Розрахунок вагомості параметрів

ПараметриX_i	ПараметриX_j				Перша ітер.		Друга ітер.		Третя ітер	
	X1	X2	X3	X4	b_i	K	b_i^1	$K_{\text{вi}}^1$	b_i^2	$K_{\text{вi}}^2$
X1	1	0,5	0,5	1,5	3,5	0,219	12,25	0,208	44,875	0,208
X2	1,5	1	1,5	1,5	5,5	0,344	21,25	0,36	77,875	0,361
X3	1,5	0,5	1	1,5	4,5	0,281	16,25	0,275	59,125	0,274
X4	0,5	0,5	0,5	1	2,5	0,156	9,25	0,157	34,125	0,157
Всього:					16	1	59	1	216	1

4.4 Аналіз рівня якості варіантів реалізації функцій

Визначаємо рівень якості кожного варіанту виконання основних функцій окремо.

Абсолютні значення параметрів Х2(кількість користувачів платформи) та Х1 (швидкодія мови програмування) відповідають технічним вимогам умов функціонування даного ПП.

Абсолютне значення параметра Х3 (кросплатформенність) обрано не найгіршим (не мінімальним), тобто це значення відповідає або варіанту а) 50% або варіанту б)100%.

Коефіцієнт технічного рівня для кожного варіанта реалізації ПП розраховується так (таблиця 4.6):

$$K_K(j) = \sum_{i=1}^{n} K_{\text{в}i,j} B_{i,j},$$

де n – кількість параметрів;

$K_{\text{в}i}$– коефіцієнт вагомості i–го параметра;

B_i – оцінка i–го параметра в балах.

Таблиця 4.6 – Розрахунок показників рівня якості варіантів реалізації основних функцій ПП

Основні функції	Варіант реалізації функції	Абсолютне значення параметра	Бальна оцінка параметра	Коефіцієнт вагомості параметра	Коефіцієнт рівня якості
F1(X1)	А	7000	1,7	0,208	0,354
F2(X2)	А	16	3,6	0,361	1,299
F3(X3,X4)	А	50	3,3	0,274	0,904
	Б	100	1,4	0,157	0,219

За даними з таблиці 4.6 за формулою:

$$K_K = K_{\text{ТУ}}[F_{1k}] + K_{\text{ТУ}}[F_{2k}] + \ldots + K_{\text{ТУ}}[F_{zk}],$$

визначаємо рівень якості кожного з варіантів:

$$K_{K1} = 0{,}354 + 1{,}299 + 0{,}904 = 2.557$$

$$K_{K2} = 0{,}354 + 1{,}299 + 0{,}219 = 1.872$$

Як видно з розрахунків, кращим є перший варіант, а основним фактором, який вплинув на результат стала популярність платформи Windows серед користувачів та простота WPF.

4.5 Економічний аналіз варіантів розробки ПП

Для визначення вартості розробки ПП спочатку проведемо розрахунок трудомісткості.

Всі варіанти включають в себе два окремих завдання:

1. Розробка проекту програмного продукту;

2. Розробка програмної оболонки;

Завдання 1 за ступенем новизни відноситься до групи А, завдання 2 – до групи Б. За складністю алгоритми, які використовуються в завданні 1 належать до групи 1; а в завданні 2 – до групи 3.

Для реалізації завдання 1 використовується довідкова інформація, а завдання 2 використовує інформацію у вигляді даних.

Проведемо розрахунок норм часу на розробку та програмування для кожного з завдань.

Проведемо розрахунок норм часу на розробку та програмування для кожного з завдань. Загальна трудомісткість обчислюється як

$$T_O = T_P \cdot K_П \cdot K_{СК} \cdot K_M \cdot K_{СТ} \cdot K_{СТ.М}$$

де T_P – трудомісткість розробки ПП;

$K_П$ – поправочний коефіцієнт;

$K_{СК}$ – коефіцієнт на складність вхідної інформації;

K_M – коефіцієнт рівня мови програмування;

$К_{СТ}$ – коефіцієнт використання стандартних модулів і прикладних програм;

$К_{СТ.М}$ – коефіцієнт стандартного математичного забезпечення.

Для першого завдання, виходячи із норм часу для завдань розрахункового характеру степеню новизни А та групи складності алгоритму 1, трудомісткість дорівнює: $Т_Р$ =80 людино-днів. Поправочний коефіцієнт, який враховує вид нормативно-довідкової інформації для першого завдання: $К_П$ = 1.7. Поправочний коефіцієнт, який враховує складність контролю вхідної та вихідної інформації для всіх семи завдань рівний 1: $К_{СК}$ = 1. Оскільки при розробці першого завдання використовуються стандартні модулі, врахуємо це за допомогою коефіцієнта $К_{СТ}$ = 0.8. Тоді, за формулою 5.1, загальна трудомісткість програмування першого завдання дорівнює:

$$Т_1 = 80 \cdot 1.7 \cdot 0.8 = 108.8 \text{ людино-днів.}$$

Проведемо аналогічні розрахунки для подальших завдань.

Для другого завдання (використовується алгоритм третьої групи складності, степінь новизни Б), тобто $Т_Р$ =30 людино-днів, $К_П$ =0.9, $К_{СК}$ = 1, $К_{СТ}$ =0.8:

$$Т_2 = 30 \cdot 0.9 \cdot 0.8 = 21.6 \text{ людино-днів.}$$

Складаємо трудомісткість відповідних завдань для кожного з обраних варіантів реалізації програми, щоб отримати їх трудомісткість:

$$Т_I = (108.8 + 21.6 + 4.8 + 21.6) \cdot 8 = 1\ 254.4 \text{ людино-годин;}$$

$$T_{II} = (108.8 + 21.6 + 6.91 + 21.6) \cdot 8 = 1\ 271.28 \text{ людино-годин;}$$

Найбільш високу трудомісткість має варіант II.

В розробці беруть участь програміст з окладом 25 000 грн. і фінансовий аналітик з окладом 20 000 грн. Визначимо середню зарплату за годину за формулою:

$$\text{СЧ} = \frac{\text{М}}{T_m \cdot t \cdot n} \text{грн.,}$$

де М – місячний оклад працівників;

T_m – кількість робочих днів на місяць;

t – кількість робочих годин на день;

n – кількість працівників.

$$\text{СЧ} = \frac{45000}{21 \cdot 8 \cdot 2} = 133.93\text{грн.}$$

Тоді, розрахуємо заробітну плату за формулою:

$$\text{СЗП} = \text{С}_\text{ч} \cdot \text{Т}_i \cdot \text{КД,}$$

де $С_ч$– величина погодинної оплати праці програміста;

$Т_i$ – трудомісткість відповідного завдання;

$К_Д$ – норматив, який враховує додаткову заробітну плату.

Зарплата розробників за варіантами становить:

I. $С_{ЗП} = 133.93 \cdot 1\ 254.4 \cdot 1.2 = 201\ 602.15$ грн.

II. $С_{ЗП} = 133.93 \cdot 1\ 271.28 \cdot 1.2 = 204\ 315.04$ грн.

Відрахування на єдиний соціальний внесок в залежності від групи професійного ризику (II клас) становить 22%:

I. $С_{ВІД} = С_{ЗП} \cdot 0.22 = 201\ 602.15 \cdot 0.22 = 44\ 352.47$ грн.

II. $С_{ВІД} = С_{ЗП} \cdot 0.22 = 204\ 315.04 \cdot 0.22 = 44\ 949.31$ грн.

Визначимо витрати на оплату однієї машино-години. ($С_М$)

Так як одна ЕОМ обслуговує одного програміста з окладом 25 000 грн., з коефіцієнтом зайнятості 0,3 то для однієї машини отримаємо:

$$С_Г = 12 \cdot М \cdot К_З = 12 \cdot 25000 \cdot 0,3 = 90\ 000 \text{ грн.}$$

З урахуванням додаткової заробітної плати:

$$С_{ЗП} = С_Г \cdot (1 + К_З) = 90\ 000 \cdot (1 + 0.3) = 117\ 000 \text{ грн.}$$

Відрахування на єдиний соціальний внесок:

$$С_{ВІД} = С_{ЗП} \cdot 0.22 = 117\ 000 \cdot 0,22 = 25\ 740 \text{ грн.}$$

Амортизаційні відрахування розраховуємо при амортизації 25% та вартості ЕОМ – 30 000 грн.

$$С_А = К_{ТМ} \cdot К_А \cdot Ц_{ПР} = 1.15 \cdot 0.25 \cdot 30\,000 = 8\,625 \text{ грн.},$$

де $К_{ТМ}$– коефіцієнт, який враховує витрати на транспортування та монтаж приладу у користувача;

$К_А$– річна норма амортизації;

$Ц_{ПР}$– договірна ціна приладу.

Витрати на ремонт та профілактику розраховуємо як:

$$С_Р = К_{ТМ} \cdot Ц_{ПР} \cdot К_Р = 1.15 \cdot 30\,000 \cdot 0.05 = 1\,725 \text{ грн.},$$

де $К_Р$– відсоток витрат на поточні ремонти.

Ефективний годинний фонд часу ПК за рік розраховуємо за формулою:

$$Т_{ЕФ} = (Д_К – Д_В – Д_С – Д_Р) \cdot t_З \cdot К_В = (365 – 104 – 8 – 16) \cdot 8 \cdot 0.6 = 1\,137.6 \text{ годин},$$

де $Д_К$ – календарна кількість днів у році;

$Д_В$, $Д_С$ – відповідно кількість вихідних та святкових днів;

$Д_Р$ – кількість днів планових ремонтів устаткування;

t –кількість робочих годин в день;

$К_В$– коефіцієнт використання приладу у часі протягом зміни.

Витрати на оплату електроенергії розраховуємо за формулою:

$$С_{ЕЛ} = Т_{ЕФ} \cdot N_С \cdot К_З \cdot Ц_{ЕН} = 1137,6 \cdot 0,156 \cdot 0,3 \cdot 2,7515 = 146,49 \text{ грн.},$$

де $N_С$ – середньо-споживча потужність приладу;

$К_З$– коефіцієнтом зайнятості приладу;

$Ц_{ЕН}$ – тариф за 1 КВт-годин електроенергії – 2,7515.

Накладні витрати розраховуємо за формулою:

$$С_Н = Ц_{ПР} \cdot 0.67 = 30\,000 \cdot 0,67 = 20\,100 \text{ грн.}$$

Тоді, річні експлуатаційні витрати будуть:

$$С_{ЕКС} = С_{ЗП} + С_{ВІД} + С_А + С_Р + С_{ЕЛ} + С_Н$$

$$С_{ЕКС} = 117\,000 + 25\,740 + 8\,625 + 1\,725 + 146.49 + 20\,100 = 173\,336.49 \text{ грн.}$$

Собівартість однієї машино-години ЕОМ дорівнюватиме:

$$С_{М\text{-}Г} = С_{ЕКС} / Т_{ЕФ} = 173\,336.49 / 1\,137.6 = 152.37 \text{ грн/год.}$$

Оскільки в даному випадку всі роботи, які пов‘язані з розробкою програмного продукту ведуться на ЕОМ, витрати на оплату машинного часу, в залежності від обраного варіанта реалізації, складає:

$$C_М = C_{М\text{-}Г} \cdot Т$$

I. $C_М = 152.37 * 1\ 254.4 = 191\ 132.93$ грн.;

II. $C_М = 152.37 * 1\ 271.28 = 193\ 704.93$ грн.;

Накладні витрати складають 67% від заробітної плати:

$$C_Н = C_{ЗП} \cdot 0.67$$

I. $C_Н = 201\ 602.15 * 0.67 = 135\ 073.44$ грн.;

II. $C_Н = 204\ 315.04 * 0.67 = 136\ 891.08$ грн.;

Отже, вартість розробки ПП за варіантами становить:

$$C_{ПП} = C_{ЗП} + C_{ВІД} + C_М + C_Н$$

I. $C_{ПП} = 201\ 602.15 + 44\ 352.47 + 191\ 132.93 + 135\ 073.44 = 572\ 160.99$ грн.;

II. $C_{ПП} = 204\ 315.04 + 44\ 949.31 + 193\ 704.93 + 136\ 891.08 = 579\ 860.36$ грн.;

4.6 Вибір кращого варіанта ПП техніко-економічного рівня

Розрахуємо коефіцієнт техніко-економічного рівня за формулою:

$$К_{ТЕРj} = К_{Кj} / С_{ппj},$$

$$К_{ТЕР1} = 2.557 / 572\ 160.99 = 4.47 \cdot 10^{-6};$$

$$К_{ТЕР2} = 1.872 / 579\ 860.36 = 3{,}23 \cdot 10^{-6};$$

Як бачимо, найбільш ефективним є першийваріант реалізації програми з коефіцієнтом техніко-економічного рівня $К_{ТЕР1} = 4.47 \cdot 10^{-6}$

4.7 Висновки до розділу4

В даному розділі проведено повний функціонально-вартісний аналіз ПП, який було розроблено в рамках дипломного проекту. Процес аналізу можна умовно розділити на дві частини.

В першій з них проведено дослідження ПП з технічної точки зору: було визначено основні функції ПП та сформовано множину варіантів їх реалізації; на основі обчислених значень параметрів, а також експертних оцінок їх важливості було обчислено коефіцієнт технічного рівня, який і дав змогу визначити оптимальну з технічної точки зору альтернативу реалізації функцій ПП.

Другу частину ФВА присвячено вибору із альтернативних варіантів реалізації найбільш економічно обґрунтованого. Порівняння запропонованих варіантів реалізації в рамках даної частини виконувалось за коефіцієнтом ефективності, для обчислення якого були обчислені такі допоміжні параметри, як трудомісткість, витрати на заробітну плату, накладні витрати.

Після виконання функціонально-вартісного аналізу програмного комплексу що розроблюється, можна зробити висновок, що з альтернатив, що залишились після першого відбору двох варіантів виконання програмного комплексу оптимальним є перший варіант реалізації

програмного продукту. У нього виявився найкращий показник техніко-економічного рівня якості $К_{ТЕР1} = 4.47 \cdot 10^{-6}$.

Цей варіант реалізації програмного продукту має такі параметри:

- Реалізація Web API – .NetCore;
- Реалізація клієнтського додатку - WPF
- цільова платформа - Windows.

Даний варіант виконання програмного комплексу дає розробникам великий простір для експериментів і вибору засобів, а користувач отримає кінцевий продукт, що задовольняє раціональні вимоги.

ВИСНОВКИ

Підводячи підсумки можна сказати, що в перших двох розділах було досліджено модель розподіленої інформаційної системи, розглянуто методи взаємодії між її елементами, проаналізовано алгоритми для синхронізації процесів.

В результаті третього розділу було розроблено приклад, що не тільки показує роботу розподілених транзакцій, а й демонструє нам просту розподілену систему. Також створено клієнтський додаток, що допомагає користувачу в експлуатації створеної системи.

В подальшому можна удосконалити систему розподілених, зробивши її незалежною від конкретних технологій (як от MySQL) та значно розширити функціонал додавши власні реалізації менеджерів даних. Паралельно можна вдосконалювати клієнтський додаток, заточуючи його під основний функціонал системи.

ЛІТЕРАТУРА

1. 1. Tanenbaum A. S. Modern operating systems. Upeer Saddle River, New Jersey: Prentice-Hall, 2015. 1038 p.

2. Коваленко А.Є. Операційні системи: навч. посібн. Київ: НТУУ «КПІ», 2010. 248с.

3. Коваленко А.Є. Розподілені інформаційні системи: навч. посібн. Київ: НТУУ «КПІ», 2008. 244с.

4. Stallings W. Operating systems: internals and design principles. Upeer Saddle River, New Jersey: Prentice-Hall, 2015. 800 p.

5. Silberschats A., Galvin P.B., Gagne G. Operating system concepts. Jefferson City: Wileys &Sons, 2012. 944 p.

6. Buchanan l.R., Linowes R.O. Making distributed data processing work. Harvard Business Review, 1980. 58(5) p.143-161.

7. Hammer M., Champy J. Re-engineering the Corporation. New Jersey: Prentice-Hall, 1993. 223 p.

8. Lucas H. Jr. The Analysis, Design, and Implementation of Information Systems. McGraw-Hill, Inc, USA, 1992. 550 p.

9. Martin J. Information Engineering, Book 3. Upeer Saddle River, New Jersey: Prentice-Hall, 1989. 542 P.

10. Terplan K., Huntington-Lee J. Distributed Systems and Netwrok Management. New York, USA: Van Nostrand Reinhold, 1995. 390 P.

11. Vigna P, Casey MJ. The Age of Cryptocurrency: How Bitcoin and the Blockchain Are Challenging the Global Economic Order. St. Martin's Press, 2015. 120 p.

12. Hieu., Vu, Quang. Peer-to-peer computing: principles and applications. Heidelberg, 1961. 16 p.

13. LeLann, G. Distributed systems - toward a formal approach. Information Processing, 1977. P. 155-160.

14. Korach E., Kutten S., Moran S. A Modular Technique for the Design of Efficient Distributed Leader Finding Algorithms. ACM Transactions on Programming Languages and Systems, 1990. P. 84–101.

Додаток А

Рисунок А1 – Слайд №1

Рисунок А2 – Слайд №2

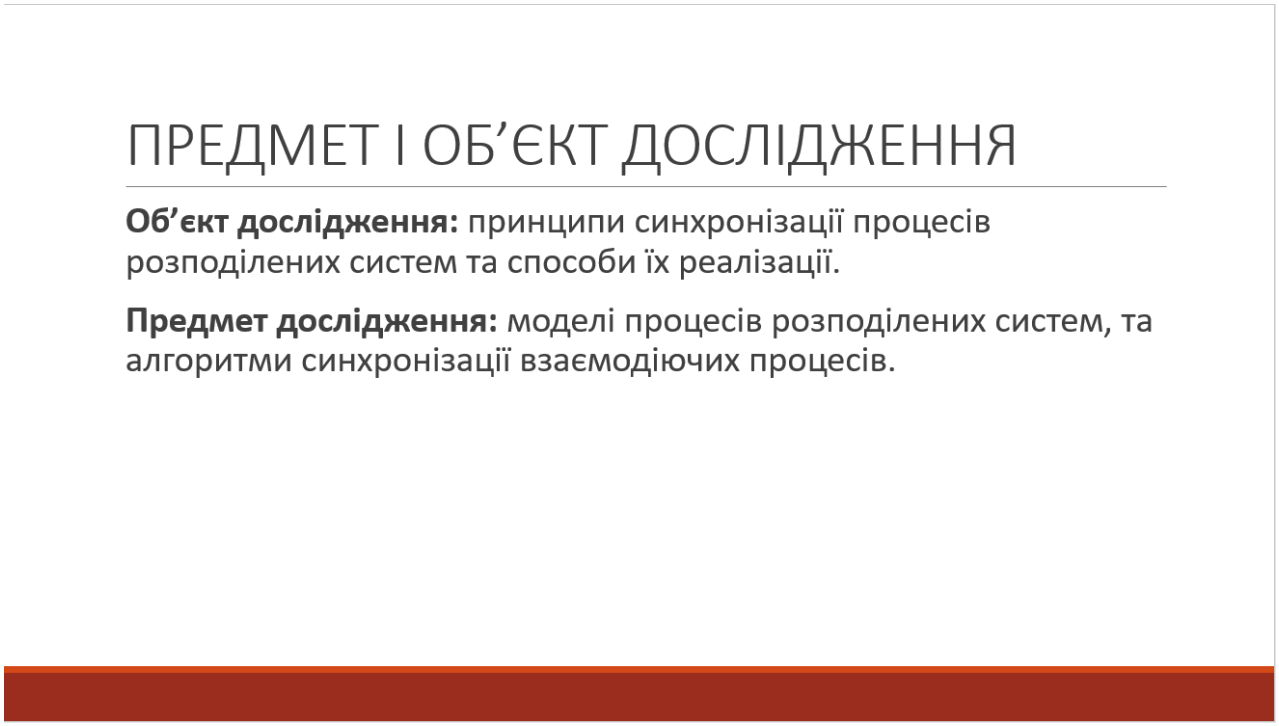


Рисунок А3 – Слайд№3

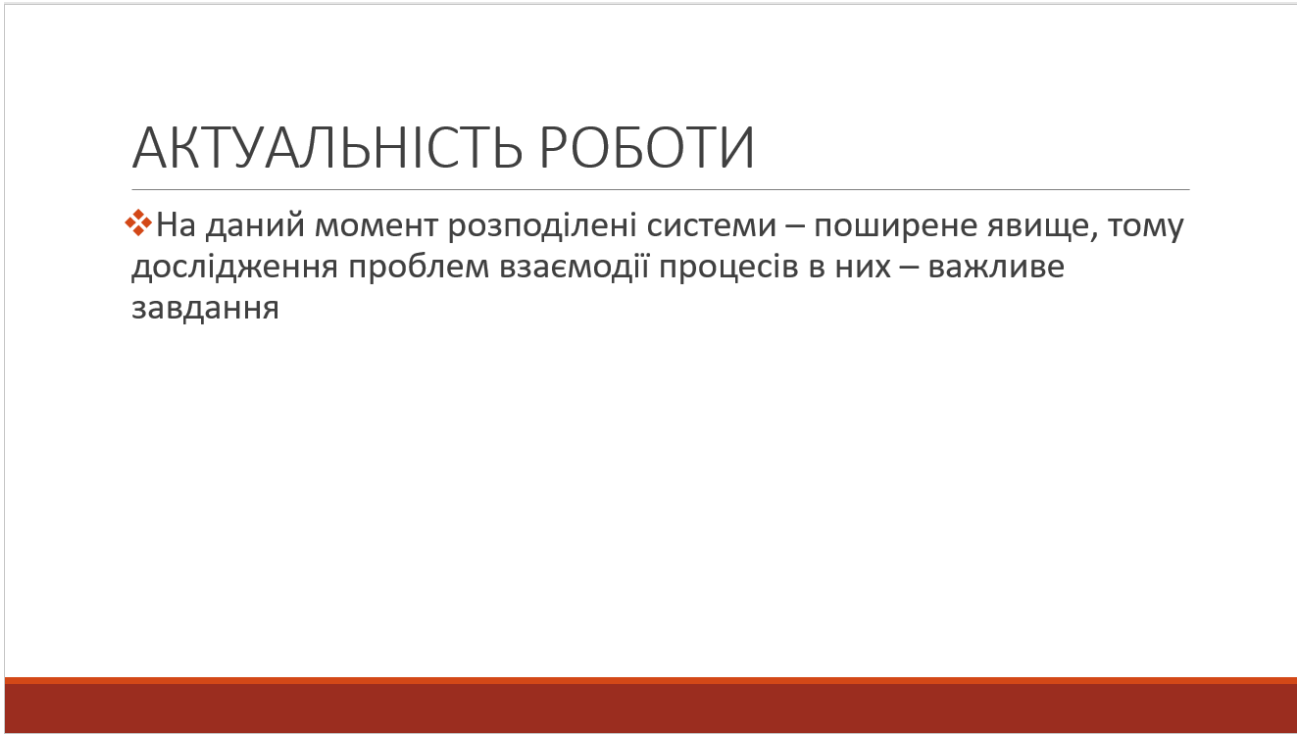


Рисунок А4 – Слайд №4

ПОНЯТТЯ РОЗПОДІЛЕНОЇ СИСТЕМИ

Розподілена система – це комплекс незалежних комп'ютерів, які сприймаються користувачем єдиною об'єднаною системою. Розподілена інформаційна система (РІС) є розподіленою системою з єдиними інформаційними ресурсами.

Доступ до інформаційних ресурсів регламентується встановленими правилами доступу до неї, процесами обробки, передавання, зберігання, перетворення й використання інформації.

Рисунок А5 – Слайд №5

СХЕМА РОЗПОДІЛЕНОЇ СИСТЕМИ

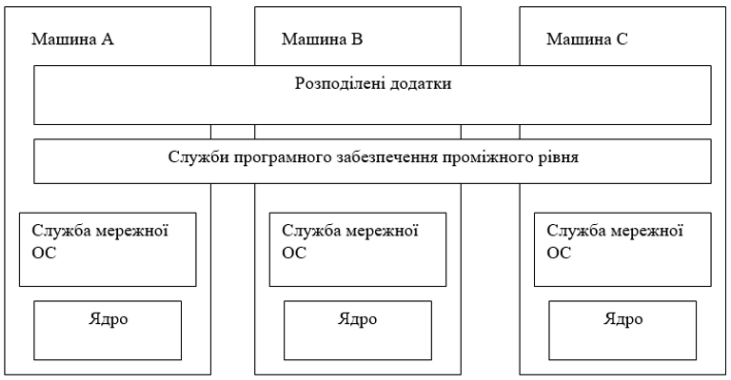


Рисунок А6 – Слайд №6

Рисунок А7 – Слайд №7

Рисунок А8 – Слайд №8

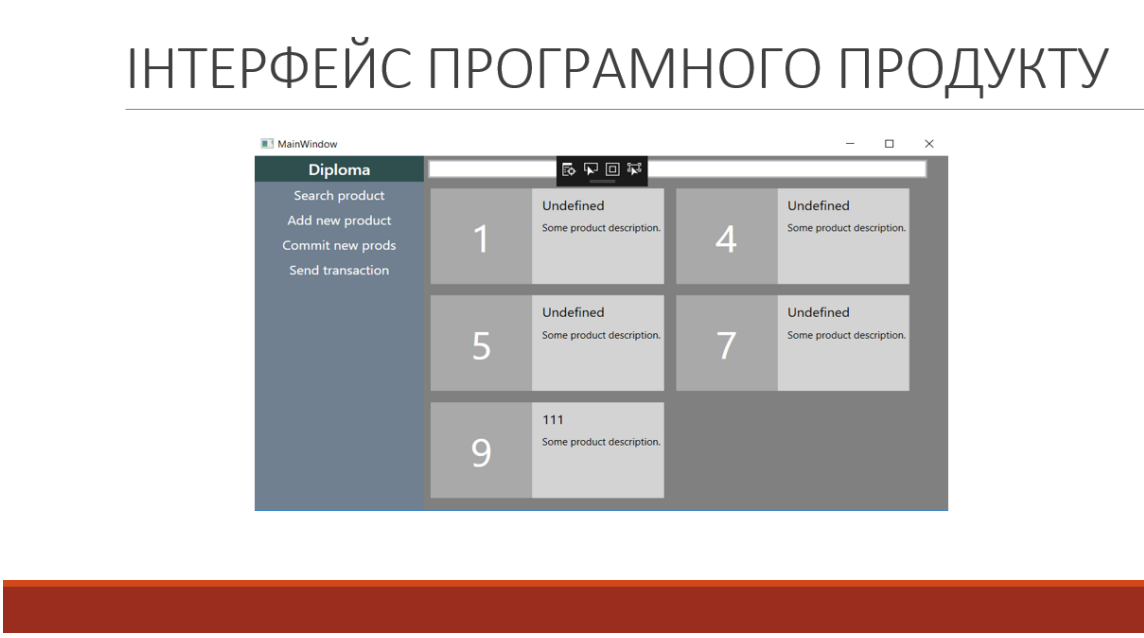


Рисунок А9 – Слайд №9

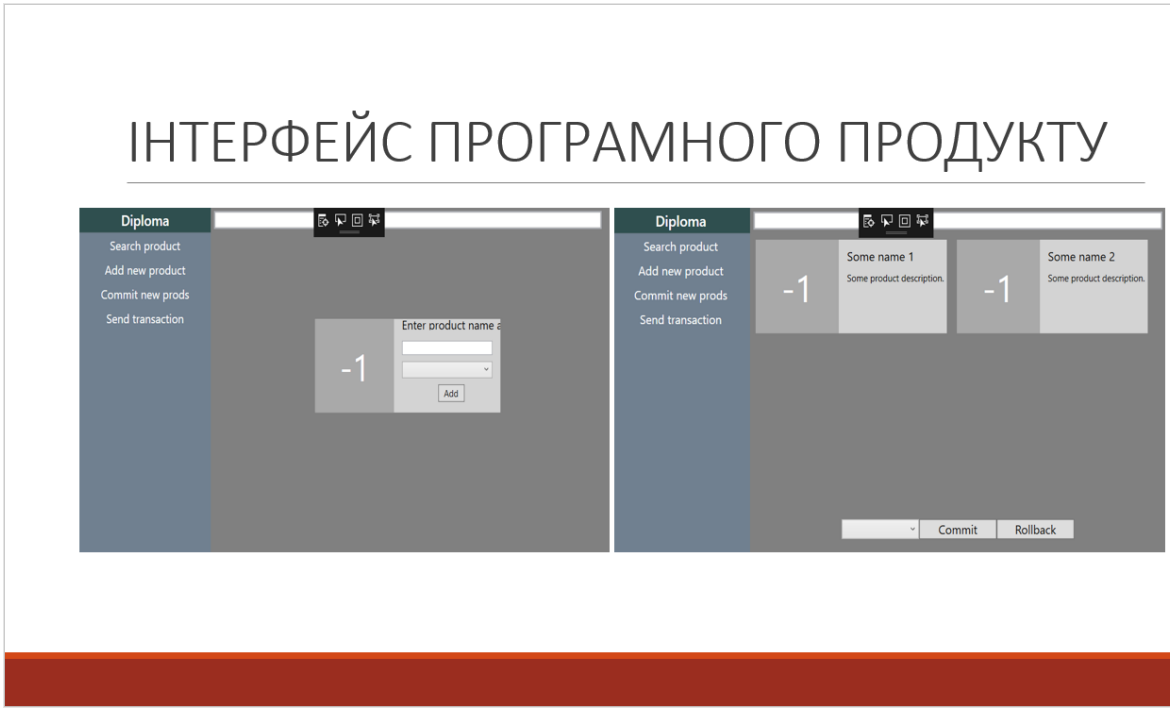


Рисунок А10 – Слайд №10

ВИСНОВКИ

- Досліджено модель розподіленої інформаційної системи
- Розглянуто методи взаємодії між її елементами
- Проаналізовано алгоритми для синхронізації процесів
- Розроблено приклад, що не тільки показує роботу розподілених транзакцій, а й демонструє просту розподілену систему
- Створити клієнтський додаток, що допомагає користувачу в експлуатації створеної системи

Рисунок А11 – Слайд №11

ПЕРСПЕКТИВИ ДОСЛІДЖЕННЯ

- Удосконалити систему розподілених, зробивши її незалежною від конкретних технологій
- Розширити функціонал додавши власні реалізації менеджерів даних
- Вдосконалювати клієнтський додаток, заточуючи його під основний функціонал системи

Рисунок А12 – Слайд №12

Додаток Б

```
<Windowx:Class="Client.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
xmlns:local="clr-namespace:Client"
mc:Ignorable="d"
Title="MainWindow" Height="450" Width="820">
<Grid  Background="Gray">
<Grid.RowDefinitions>
<RowDefinition Height="Auto"/>
<RowDefinition Height="*"/>
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
<ColumnDefinition Width="200"></ColumnDefinition>
<ColumnDefinition Width="600"></ColumnDefinition>
</Grid.ColumnDefinitions>
<Grid Background="DarkSlateGray" Grid.Column="0" Grid.Row="0">
<TextBlock Text="Diploma" HorizontalAlignment="Center" FontSize="19"
Foreground="White" FontWeight="DemiBold" VerticalAlignment="Center"></TextBlock>
</Grid>
<Grid Background="SlateGray" Grid.Column="0" Grid.Row="1" Grid.RowSpan="2">
<Grid.RowDefinitions>
<RowDefinition Height="Auto"></RowDefinition>
<RowDefinition Height="Auto"></RowDefinition>
```

```
<RowDefinition Height="Auto"></RowDefinition>

<RowDefinition Height="Auto"></RowDefinition>

<RowDefinition Height="*"></RowDefinition>

</Grid.RowDefinitions>

<TextBlock HorizontalAlignment="Stretch" TextAlignment="Center"
VerticalAlignment="Stretch" MouseDown="TextBlock_MouseDown" Grid.Row="0"
Foreground="White" FontSize="16" Margin="5" Text="Search product">

<TextBlock.Style>

<Style TargetType="TextBlock">

<Style.Triggers>

<Trigger Property="IsMouseOver" Value="True">

<Setter Property="Foreground" Value="LightGray"></Setter>

</Trigger>

</Style.Triggers>

</Style>

</TextBlock.Style>

</TextBlock>

<TextBlock HorizontalAlignment="Center" MouseDown="TextBlock_MouseDown_1"
Grid.Row="1" Foreground="White" FontSize="16" Margin="5" Text="Add new
product"></TextBlock>

<TextBlock HorizontalAlignment="Center" MouseDown="TextBlock_MouseDown_2"
Grid.Row="2" Foreground="White" FontSize="16" Margin="5" Text="Commit new
prods"></TextBlock>

<TextBlock HorizontalAlignment="Center" MouseDown="TextBlock_MouseDown_3"
Grid.Row="3" Foreground="White" FontSize="16" Margin="5" Text="Send
transaction"></TextBlock>

</Grid>

<Grid Margin="5" Grid.Row="0" Grid.Column="1">

<Grid.ColumnDefinitions>
```

```
<ColumnDefinition Width="*"></ColumnDefinition>
<ColumnDefinition Width="Auto"></ColumnDefinition>
</Grid.ColumnDefinitions>
<TextBox Grid.Column="0" BorderThickness="2" FontSize="14"
TextChanged="TextBox_TextChanged"/>
<Button Content="Refresh" Grid.Column="1" Click="Button_Click_3"></Button>
</Grid>
<ListBox ScrollViewer.HorizontalScrollBarVisibility="Disabled" Width="600"
Grid.Column="1" BorderThickness="0" Background="Gray" Visibility="{Binding
SearchBlockVisibility}" ItemsSource="{Binding ItemSource}" Grid.Row="1"
SelectionChanged="ListBox_SelectionChanged">
<ListBox.ItemsPanel>
<ItemsPanelTemplate>
<WrapPanel IsItemsHost="True" ItemWidth="290"/>
</ItemsPanelTemplate>
</ListBox.ItemsPanel>
<ListBox.ItemContainerStyle>
<Style TargetType="ListBoxItem">
<Setter Property="Padding" Value="0"/>
<Setter Property="Margin" Value="6"/>
</Style>
</ListBox.ItemContainerStyle>
<ListBox.ItemTemplate>
<DataTemplate>
<Grid Width="282" Background="LightGray">
<Grid.ColumnDefinitions>
<ColumnDefinition Width="132*"/>
<ColumnDefinition Width="150*"/>
```

```
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
<RowDefinition/>
<RowDefinition Height="40"/>
<RowDefinition/>
</Grid.RowDefinitions>
<Grid Grid.RowSpan="3" Margin="0,0,12,0" Background="DarkGray" Width="120"
Height="120" HorizontalAlignment="Left">
<TextBlock Text="{Binding Id}" HorizontalAlignment="Center" VerticalAlignment="Center"
FontSize="48" Foreground="White"/>
</Grid>
<TextBlock Grid.Column="1" Grid.ColumnSpan="3" Text="{Binding Name}" FontSize="16"
VerticalAlignment="Center"/>
<TextBlock Grid.Row="1" Grid.Column="1" Grid.ColumnSpan="3" Text="Some product
description." TextWrapping="Wrap"/>
</Grid>
</DataTemplate>
</ListBox.ItemTemplate>
</ListBox>
<Grid Margin="20,-50,20,20" Grid.Column="1" Visibility="{Binding AddBlockVisibility}"
Grid.Row="1" Height="120">
<Grid Width="282" Background="LightGray">
<Grid.ColumnDefinitions>
<ColumnDefinition Width="132*"/>
<ColumnDefinition Width="150*"/>
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
<RowDefinition/>
```

```
<RowDefinition/>
<RowDefinition Height="Auto"/>
<RowDefinition Height="Auto"/>
<RowDefinition Height="Auto"/>
<RowDefinition/>
</Grid.RowDefinitions>
<Grid Grid.RowSpan="6" Margin="0,0,12,0" Background="DarkGray" Width="120" Height="120" HorizontalAlignment="Left">
<TextBlock Text="{Binding InputedProduct.Id}" HorizontalAlignment="Center" VerticalAlignment="Center" FontSize="48" Foreground="White"/>
</Grid>
<TextBlock Grid.Column="1" Grid.ColumnSpan="3" Text="Enter product name and branch" FontSize="16" VerticalAlignment="Center"/>
<TextBox Text="{Binding InputedProduct.Name, Mode=TwoWay}" Grid.Row="2" Grid.Column="1" Grid.ColumnSpan="3" Margin="0,0,12,0"/>
<Button Width="40" Click="Button_Click" Content="Add" Grid.Row="4" Grid.Column="1" Grid.ColumnSpan="3" Margin ="0,8,0,0" Height="22" HorizontalAlignment="Center" VerticalAlignment="Center"></Button>
<ComboBox Grid.Row="3" Grid.Column="1" SelectedValuePath="Content" SelectedValue="{Binding SelectedBranch}" Grid.ColumnSpan="3" Margin ="0,8,12,0">
<ComboBoxItem Content="Kiev"></ComboBoxItem>
<ComboBoxItem Content="Odesa"></ComboBoxItem>
</ComboBox>
</Grid>
</Grid>
<Grid Grid.Column="1" Grid.Row="1" Visibility="{Binding CommitBlockVisibility}" Background="Gray">
<Grid.RowDefinitions>
<RowDefinition Height="*"/>
```

```
<RowDefinition Height="Auto"/>
</Grid.RowDefinitions>
<ListBox Grid.Row="0" ScrollViewer.HorizontalScrollBarVisibility="Disabled" Width="600"
Background="Gray" BorderThickness="0" ItemsSource="{Binding AddedItems,
Mode=TwoWay}">
<ListBox.ItemsPanel>
<ItemsPanelTemplate>
<WrapPanel IsItemsHost="True" ItemWidth="290" />
</ItemsPanelTemplate>
</ListBox.ItemsPanel>
<ListBox.ItemContainerStyle>
<Style TargetType="ListBoxItem">
<Setter Property="Padding" Value="0"/>
<Setter Property="Margin" Value="6"/>
</Style>
</ListBox.ItemContainerStyle>
<ListBox.ItemTemplate>
<DataTemplate>
<Grid Width="282" Background="LightGray">
<Grid.ColumnDefinitions>
<ColumnDefinition Width="132*"/>
<ColumnDefinition Width="150*"/>
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
<RowDefinition/>
<RowDefinition Height="40"/>
<RowDefinition/>
</Grid.RowDefinitions>
```

```
<Grid Grid.RowSpan="3" Margin="0,0,12,0" Background="DarkGray" Width="120"
Height="120" HorizontalAlignment="Left">
<TextBlock Text="{Binding Id}" HorizontalAlignment="Center" VerticalAlignment="Center"
FontSize="48" Foreground="White"/>
</Grid>
<TextBlock Grid.Column="1" Grid.ColumnSpan="3" Text="{Binding Name}" FontSize="16"
VerticalAlignment="Center"/>
<TextBlock Grid.Row="1" Grid.Column="1" Grid.ColumnSpan="3" Text="Some product
description." TextWrapping="Wrap"/>
</Grid>
</DataTemplate>
</ListBox.ItemTemplate>
</ListBox>
<Grid Margin="20" Grid.Row="1">
<Grid.ColumnDefinitions>
<ColumnDefinition Width="*"/>
<ColumnDefinition Width="*"/>
<ColumnDefinition Width="*"/>
<ColumnDefinition Width="*"/>
<ColumnDefinition Width="*"/>
</Grid.ColumnDefinitions>
<ComboBox Grid.Column="1" SelectedValuePath="Content" SelectedValue ="{Binding
SelectedBranch}">
<ComboBoxItem Content="Kiev"></ComboBoxItem>
<ComboBoxItem Content="Odesa"></ComboBoxItem>
</ComboBox>
<Button Grid.Column="2" Click="Button_Click_1" Content="Commit"
FontSize="16"></Button>
```

Додаток В

```
using Newtonsoft.Json;
using Objects;
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.Linq;
using System.Net.Http;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using Action = Objects.Action;

namespace Client
{
public partial class MainWindow : Window, INotifyPropertyChanged
```

```
{
public MainWindow()
{
InitializeComponent();
this.DataContext = this;
SearchBlockVisibility = Visibility.Visible;
InputedProduct = new Product() { Id = -1, Name = "" };
_allAdedItems = new ObservableCollection<Product>();
_addedItems = new ObservableCollection<Product>();
_actions = new List<Action>();
Task.Run(async () => _allToShow = ItemSource =
(JsonConvert.DeserializeObject(await new
HttpClient().GetStringAsync("https://localhost:44303/TransactionManager"),
typeof(IEnumerable<Product>)) as IEnumerable<Product>).ToList()).Wait();
}



private IList<Action> _actions;


private IList<Product> _allToShow;
private IList<Product> _itemSource;
public IList<Product> ItemSource
{
get
{
return _itemSource;
```

```
}
set
{
_itemSource = value;
PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("ItemSource"));
}
}

public event PropertyChangedEventHandler PropertyChanged;

private void ListBox_SelectionChanged(object sender, SelectionChangedEventArgs e)
{

}

private void TextBox_TextChanged(object sender, TextChangedEventArgs e)
{
ItemSource = _allToShow.Where(x => x.Name.Contains((sender as TextBox).Text)).ToList();
AddedItems = new ObservableCollection<Product>(_allAdedItems.Where(x =>
x.Name.Contains((sender as TextBox).Text)));
}

private Visibility _addBlockVisibility;
public Visibility AddBlockVisibility
{
get
{
```

```
return _addBlockVisibility;
}
set
{
_addBlockVisibility = value;
if (value == Visibility.Visible)
{
SearchBlockVisibility = Visibility.Collapsed;
CommitBlockVisibility = Visibility.Collapsed;
}
PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("AddBlockVisibility"));
}
}

private Visibility _searchBlockVisibility;
public Visibility SearchBlockVisibility
{
get
{
return _searchBlockVisibility;
}
set
{
_searchBlockVisibility = value;
if (value == Visibility.Visible)
{
AddBlockVisibility = Visibility.Collapsed;
```

```
CommitBlockVisibility = Visibility.Collapsed;
}
PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("SearchBlockVisibility"));
}
}

private Visibility _commitBlockVisibility;
public Visibility CommitBlockVisibility
{
get
{
return _commitBlockVisibility;
}
set
{
_commitBlockVisibility = value;
if (value == Visibility.Visible)
{
AddBlockVisibility = Visibility.Collapsed;
SearchBlockVisibility = Visibility.Collapsed;
}
PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("CommitBlockVisibility"));
}
}

private Product _inputedProduct;
public Product InputedProduct
```

```
{
get { return _inputedProduct; }
set
{
_inputedProduct = value;
PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("InputedProduct"));
}
}

private ObservableCollection<Product> _addedItems;
private ObservableCollection<Product> _allAdedItems;
public ObservableCollection<Product> AddedItems
{
get { return _addedItems; }
set
{
_addedItems = value;
PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("AddedItems"));
}
}

private string _selectedBranch;
public string SelectedBranch
{
get { return _selectedBranch; }
set
{
```

```
_selectedBranch = value;
PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("SelectedBranch"));
}
}

private void TextBlock_MouseDown(object sender, MouseButtonEventArgs e)
{
SearchBlockVisibility = Visibility.Visible;
}

private void TextBlock_MouseDown_1(object sender, MouseButtonEventArgs e)
{
AddBlockVisibility = Visibility.Visible;
}

private void TextBlock_MouseDown_2(object sender, MouseButtonEventArgs e)
{
CommitBlockVisibility = Visibility.Visible;
}

private void Button_Click(object sender, RoutedEventArgs e)
{
_allAdedItems.Add(InputedProduct);
AddedItems.Add(InputedProduct);
_actions.Add(new Action() { Branch = SelectedBranch, CurrentObject =
JsonConvert.SerializeObject(InputedProduct), Type = "ins" });
InputedProduct = new Product() { Id = -1, Name = "" };
```

```
}

private void Button_Click_1(object sender, RoutedEventArgs e)
{
_actions.Add(new Action() { Branch = SelectedBranch, CurrentObject =
JsonConvert.SerializeObject(InputedProduct), Type = "save" });
_allAdedItems.Clear();
AddedItems.Clear();
}

private void Button_Click_2(object sender, RoutedEventArgs e)
{
_actions.Add(new Action() { Branch = SelectedBranch, CurrentObject =
JsonConvert.SerializeObject(InputedProduct), Type = "rollback" });
_allAdedItems.Clear();
AddedItems.Clear();
}

private void TextBlock_MouseDown_3(object sender, MouseButtonEventArgs e)
{
if (_actions.Count == 0)
return;
if (!_actions.Any(_ => _.Type == "save" || _.Type == "rollback"))
if (MessageBox.Show("Are you shure that you want to send transaction withot save or
rollback?", "Warning", MessageBoxButton.YesNo, MessageBoxImage.Warning) ==
MessageBoxResult.No)
return;
```

```
if (new HttpClient().PostAsync("https://localhost:44303/TransactionManager", new
StringContent(JsonConvert.SerializeObject(_actions).ToString(), Encoding.UTF8,
"application/json")).Result.IsSuccessStatusCode)
MessageBox.Show("Transaction done", "Info", MessageBoxButton.OK,
MessageBoxImage.Information);
}

private void Button_Click_3(object sender, RoutedEventArgs e)
{
Task.Run(async () =>
{
_allToShow = ItemSource = (JsonConvert.DeserializeObject(await new HttpClient()
.GetStringAsync("https://localhost:44303/TransactionManager"),
typeof(IEnumerable<Product>)) as IEnumerable<Product>).ToList();
}).Wait();
}
}
```

Додаток Г

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Text;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Newtonsoft.Json;

namespace TransactionManager.Controllers
{
[Route("[controller]")]
[ApiController]
public class TransactionManagerController : ControllerBase
{
[HttpGet]
public async Task<ActionResult<object>> GetAsync()
{
return await Task.Run(async () =>
{
string res = string.Empty;
foreach (var branch in branchNameToAddress)
res += await CommandToHttpRequest.GetValueOrDefault("read").Invoke(string.Empty,
branch.Key);
return res;
```

```
}
);
}

[HttpGet("{id}")]
public ActionResult<string> Get(int id)
{
return "a";
}

[HttpPost]
public async Task PostAsync([FromBody] object value)
{
await Task.Run(()=>
{
foreach (var a in (JsonConvert.DeserializeObject(value.ToString(),
typeof(IEnumerable<Objects.Action>)) as IEnumerable<Objects.Action>))
{
if (a.Type == "read")
continue;
CommandToHttpRequest.GetValueOrDefault(a.Type).Invoke(a.CurrentObject,a.Branch).Wait();
}
});
}

private static Dictionary<string, string> branchNameToAddress = new Dictionary<string,
string>()
{
```

```
{"Kiev", "https://localhost:44390/planner" },
{"Odesa", "https://localhost:44390/planner" }
};

private static Dictionary<string, Func<object,string,Task<object>>> CommandToHttpRequest =
new Dictionary<string, Func<object,string,Task<object>>>()
{
{"ins", async (content, branch)=> await (new
HttpClient()).PostAsync(branchNameToAddress[branch],new
StringContent(content.ToString(),Encoding.UTF8,"application/json"))},
{"read", async (content, branch)=> await new
HttpClient().GetStringAsync(branchNameToAddress[branch]+"/"+content.ToString())},
{"save", async (content, branch)=> await (new
HttpClient()).PostAsync(branchNameToAddress[branch] +"/save", null)},
{"rollback", async (content, branch)=> await (new
HttpClient()).PostAsync(branchNameToAddress[branch]+"/rollback", null)}
};
}
}
```

Додаток Д

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using TransactionPlanner.Models;
using Newtonsoft.Json;
using Objects;

namespace TransactionPlanner.Controllers
{
[Route("[controller]")]
[ApiController]
public class PlannerController : ControllerBase
{
public List<object> WriteQeue{get;set;}
private CorePlanner _planner;
public PlannerController(CorePlanner planner)
{
_planner = planner;
WriteQeue = new List<object>();
}

[HttpGet]
```

```
public async Task<ActionResult<IEnumerable<Product>>> GetAsync()
{
return await _planner.ReadAllAsync();
}

[HttpGet("{id}")]
public async Task<ActionResult<string>> GetAsync(int id)
{
return await _planner.ReadAsync(id);
}

[HttpPost]
public async Task PostAsync([FromBody] object value)
{
try
{
await
_planner.WriteNextProductAsync(JsonConvert.DeserializeObject(value.ToString(),typeof(Produ
ct)) as Product);
}
catch (System.Exception)
{
throw;
}
}

[HttpPost("save")]
public async Task<bool> PostAsync()
```

Додаток Е

```
using System.Collections.Generic;
using System.Data;
using System.Data.Common;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.Extensions.Configuration;
using MySql.Data.MySqlClient;
using Objects;

namespace TransactionPlanner.Models
{
public class CorePlanner
{
private IList<Product> _writeQueue;
private MySqlTransaction _currentTransaction;
private DB _db;
private IConfiguration configuration;
public CorePlanner(DB db, IConfiguration conf)
{
_db = db;
configuration = conf;
_writeQueue = new List<Product>();
_db.Connection.Open();
Product.IsFree = true;
```

```
}

public async Task WriteNextProductAsync(Product nextProd)
{
_writeQueue.Add(nextProd);
await ExecuteProduct();
}

public async Task<bool> SaveAsync()
{
return await Task<bool>.Run(()=>{
try
{
if (_writeQueue.Count>0)
{
Task a = new Task(async ()=> await ExecuteProduct());
Task.WaitAll(a);
}
_currentTransaction.Commit();
_currentTransaction.Dispose();
return true;
}
catch
{
return false;
}
});
```

```
}

public async Task<bool> RollBackAsync()
{
return await Task<bool>.Run(()=>{
try
{
if (_writeQueue.Count>0)
{
_writeQueue.Clear();
}
_currentTransaction.Rollback();
_currentTransaction.Dispose();
return true;
}
catch
{
return false;
}
});
}

public async Task<string> ReadAsync(int id)
{
return await Task.Run(()=>
{
var cmd = _db.Connection.CreateCommand();
```

```
cmd.CommandText = @"SELECT Name FROM Products1.Products WHERE idProducts =
"+id;
return cmd.ExecuteScalar().ToString();
});
}

public async Task<Product[]> ReadAllAsync()
{
return await Task.Run(async () =>
{
using (DB db = new DB(configuration["ConnectionStrings:DefaultConnection"]))
{
db.Connection.Open();
var cmd = db.Connection.CreateCommand();
cmd.CommandText = @"SELECT * FROM Products1.Products";
var tmp = new List<Product>();
var reader = await cmd.ExecuteReaderAsync();
while (await reader.ReadAsync())
{
tmp.Add(new Product() { Id = await reader.GetFieldValueAsync<int>(0), Name = await
reader.GetFieldValueAsync<string>(1) });
}
reader.Close();
db.Connection.Close();
return tmp.ToArray();
}
});
}
```

```
private async Task ExecuteProduct()
{
await Task.Run(()=>
{
var prod = _writeQueue.ElementAtOrDefault(0);

if (prod !=null &&Product.IsFree)
{
Product.IsFree = false;

if (_currentTransaction?.Connection==null)
_currentTransaction = _db.Connection.BeginTransaction();

var cmd = _db.Connection.CreateCommand();
cmd.CommandText = @"INSERT INTO `Products1`.`Products`(`idProducts`,`Name`) VALUES
("
+ ((prod.Id<0)?"null":prod.Id.ToString()) +"'" + prod.Name+ "');";
cmd.Transaction = _currentTransaction;
if (!(_db.Connection.State == ConnectionState.Executing))
{
cmd.ExecuteScalar();
_writeQueue.RemoveAt(0);
}
```